# RabbitMQ

## RabbitMQ

### RabbitMQ

RabbitMQ他是一个消息中间件，说道消息中间件【最主要的作用：信息的缓冲区】还是的从应用场景来看下:

1、系统集成与分布式系统的设计

    各种子系统通过消息来对接，这种解决方案也逐步发展成一种架构风格，即“通过消息传递的架构”。

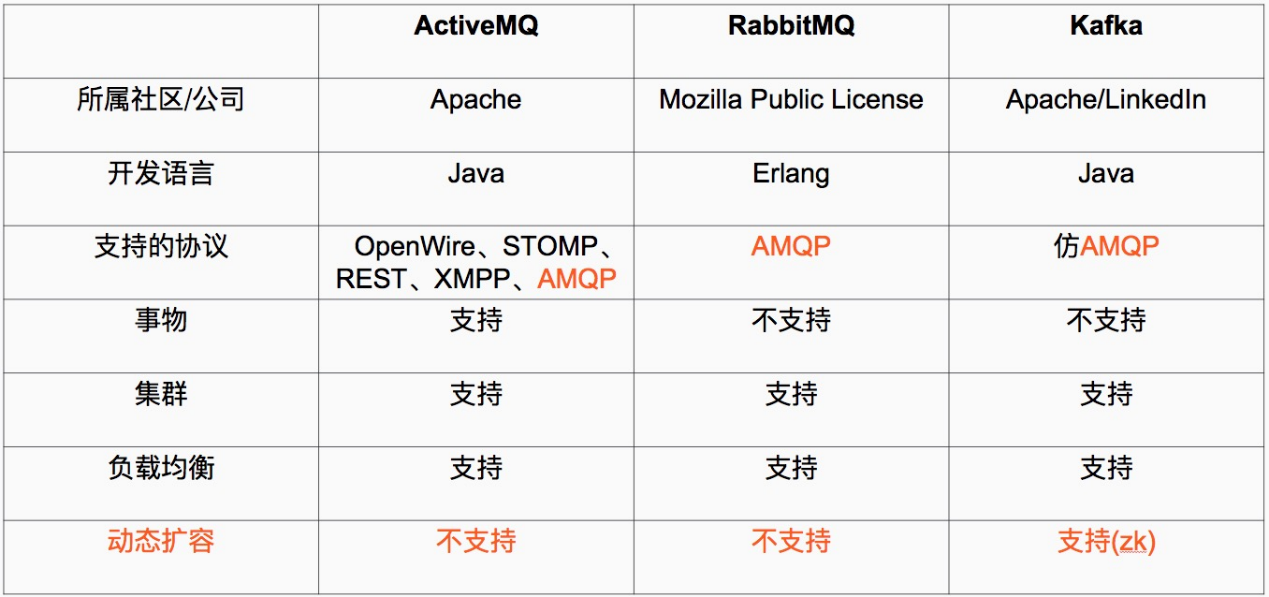
    举个例子：现在医院有两个科“看病科”和“住院科”在之前他们之间是没有任何关系的，如果你在“看病课”看完病后注册的信息和资料，到住院科后还得重新注册一遍？那现在要改革，你看完病后可以直接去住院科那两个系统之间需要打通怎么办？这里就可以使用我们的消息中间件了。

2、异步任务处理结果回调的设计

    举个例子：记录日志，假如需要记录系统中所有的用户行为日志，如果通过同步的方式记录日志势必会影响系统的响应速度，当我们将日志消息发送到消息队列，记录日志的子系统就会通过异步的方式去消费日志消息。这样不需要同步的写入日志了NICE

3、并发请求的压力高可用性设计

    举个例子：比如电商的秒杀场景。当某一时刻应用服务器或数据库服务器收到大量请求，将会出现系统宕机。如果能够将请求转发到消息队列，再由服务器去消费这些消息将会使得请求变得平稳，提高系统的可用性。



## AMQP协议

### AMQP

Advanced Message Queuing Protocol （高级消息队列协议The Advanced Message Queuing Protocol (AMQP)：是到有现代特征的二进制协议·是一个提供统一消息应用层标准高级消息队列协议,是应用层协议的一个开放标 准，为面向消息的中间件架构。

AMQP定义了通过网络发送的字节流的数据格式。因此兼容性非常好，任何实现AMQP协议的程序都可以和与AMQP协议兼容的其他程序交互，可以很容易做到跨语言，跨平台。

### AMQP的协议

AMQP协议是一个二进制协议，具有一些现代特性：多通道（multi-channel），可协商（negotiated），异步、安全、便携、语言中立、高效的。其协议主要分成两层：

**功能层（Functional Layer）**：定义了一系列的命令

**传输层（Transport Layer）**：携带了从应用 → 服务端的方法，用于处理多路复用、分帧、编码、心跳、data-representation、错误处理。

AMQ Model的设计是由以下需求驱动的：

确保符合标准的实现之间的互操作性。

提供清晰且直接的方式控制QoS

保持一致和明确的命名

通过协议能够修改服务端的各种配置

使用可以轻松映射到应用程序级API的命令符号

清晰，每个操作只能做一件事。

AMQP传输层是由以下需求驱动的

紧凑。能够快速封包和解包

可以携带任意大小的消息，没有明显的限制

同一个连接可以承载多个通道（Channel）

长时间存活，没有显著的限制

允许异步命令流水线

容易扩展。易于处理新需求、或者变更需求

向前兼容

使用强大的断言模型，可修复

对编程语言保持中立

适合代码生成过程

在设计过程中，希望能够支持不同的消息架构：

先存后发模型。有多个Writer，只有一个Reader

分散工作负载。有多个Writer和多个Reader

发布订阅模型，多个Writer和多个reader

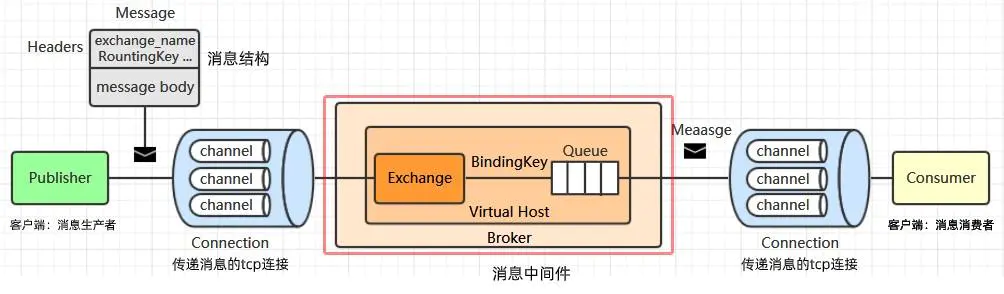
基于消息内容的路由，多个Writer，多个Reader

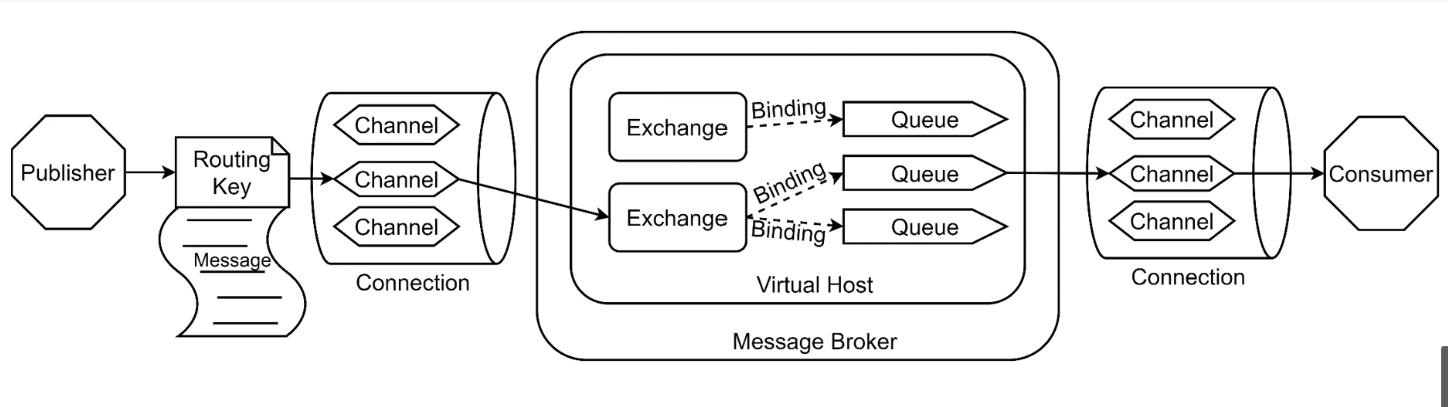
队列文件传输，多个Writer，多个Reader

两个节点之间点对点连接

市场数据（Market data）分发。多个数据源，多个Reader

### AMQP架构





Producer: 消息生产者，即投递消息的程序。

Message:消息，服务器和应用程序之间传递的数据，由Properties和Body组成。

Properties可以对消息进行修饰，比如消息的优先级、延迟等高级特性；

Body则就是消息体内容

Routing key：一个路由规则，虚拟机可用它来确定如何路由一个特定消息

Connection： 连接，应用程序与Broker的网络连接

Chanel：网络通信。几乎所有的操作都是在chanel中进行，Chanel是进行消息的读写通道，一个客户端可以建立多个Chanel，每个Chanel代表一个会话任务

Server:又称Broker，接受客户端的连接，实现AMQP实体服务

Virtual host:虚拟主机，用于进行逻辑隔离，最上层的消息路由。一个Virtual Host里面可以有若干的Exchange和Queue，同一个Virtual Host里面不能有相同名称的Exchange和Queue

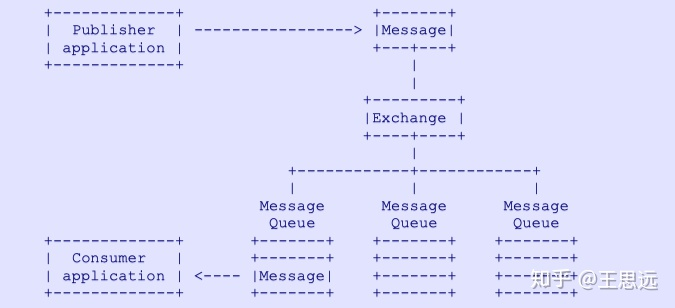
Exchange：交换机，接收消息，根据路由键转发消息到绑定的队列

Binding:Exchange和Queue之间的虚拟链接，binding中可以包含多个routing key

Queue:也称为Message Queue，消息队列，保存消息并将他们转发给消费者

Consumer: 消息消费者，即接收消息的程序。

### 消息的生命周期



1、消息由生产者产生。生产者把内容放到消息里，并设置一些属性以及消息的路由。然后生产者把消息发给服务端。

2、服务端收到消息，交换器（大部分情况）把消息路由到若干个该服务器上的消息队列中。如果这个消息找不到路由，则会丢弃或者退回给生产者（生产者可自行决定）。

3、一条消息可以存在于许多消息队列中。 服务器可以通过复制消息，引用计数等方式来实现。这不会影响互操作性。 但是，将一条消息路由到多个消息队列时，每个消息队列上的消息都是相同的。 没有可以区分各种副本的唯一标识符。

4、消息到达消息队列。消息队列会立即尝试通过AMQP将其传递给消费者。 如果做不到，消息队列将消息存储（按生产者的要求存储在内存中或磁盘上），并等待消费者准备就绪。 如果没有消费者，则消息队列可以通过AMQP将消息返回给生产者（同样，如果生产者要求这样做）。

5、当消息队列可以将消息传递给消费者时，它将消息从其内部缓冲区中删除。 可以立即删除，也可以在使用者确认其已成功处理消息之后删除(ack)。 由消费者选择“确认”消息的方式和时间。消费者也可以拒绝消息（否定确认）。

生产者发消息与消费者确认，被分组成一个事务。当一个应用同时扮演多个角色时：发消息，发ack，commit或者回滚事务。消息从服务端投递给消费者这个过程不是事务的。消费者对消息进行确认就够了。

**交换器(exchange)的生命周期**

每个AMQP服务端都会自己创建一些交换器，这些不能被销毁。AMQP程序也可以创建其自己的交换器。AMQP并不使用 create 这个方法，而是使用 declare 方法来表示：如果不存在，则创建，存在了则继续。程序可以创建交换器用于私有使用，并在任务完成后销毁它们。虽然AMQP提供了销毁交换器的方法，但一般来讲程序不需要销户它。

队列(queue)的生命周期

持久化消息队列：由很多消费者共享。当消费者都退出后，队列依然存在，并会继续收集消息。

临时消息队列：临时消息队列对于消费者是私有和绑定的。当消费者断开连接，则消息队列被删除。

### 交换机类型和属性

|  |  |
| --- | --- |
| **Name（交换机类型）** | **Default pre-declared names（预声明的默认名称）** |
| Direct exchange（直连交换机） | (Empty string) and amq.direct |
| Fanout exchange（扇型交换机） | amq.fanout |
| Topic exchange（主题交换机） | amq.topic |
| Headers exchange（头交换机） | amq.match (and amq.headers in RabbitMQ) |

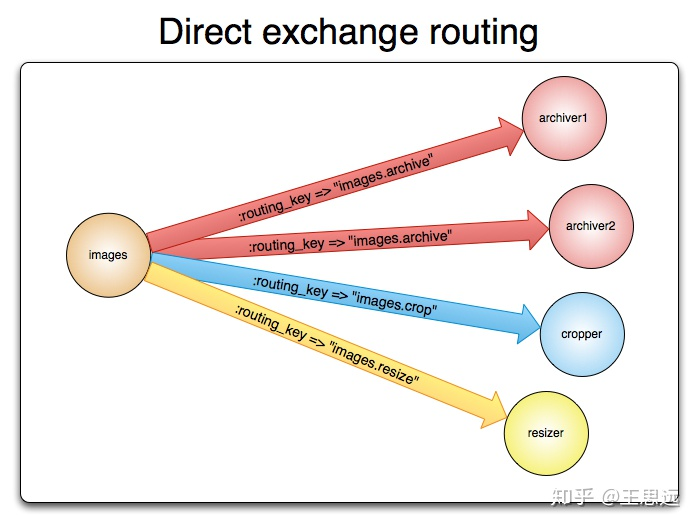
Name:交换机名称。

Durability （消息代理重启后，交换机是否还存在）。

Auto-delete （当所有与之绑定的消息队列都完成了对此交换机的使用后，删掉它）。

Arguments（依赖代理本身）。

**Direct 交换器**

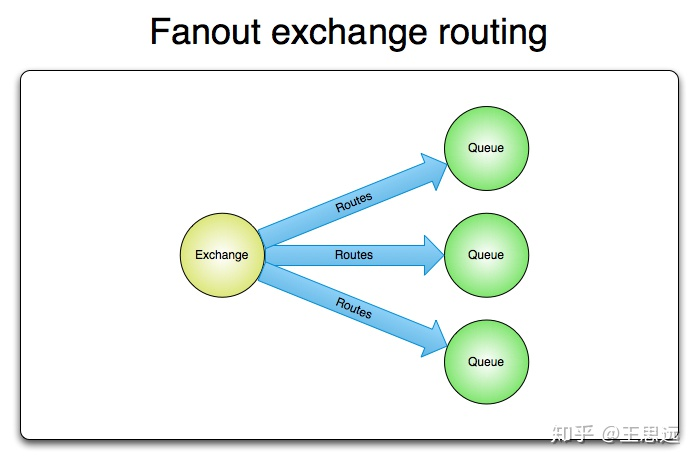


1、一个消息队列使用RoutingKey K 绑定到交换器

2、生产者向交换器发送RoutingKey为R的消息

3、当 K=R时，消息被转发到该消息队列中

**Fanout 交换器**



1、一个消息队列没有使用任何参数绑定交换器

2、生产者向交换器发了一条消息

3、这个消息无条件的发送到该消息队列

Topic 交换器

1、消息队列使用路由规则 P 绑定到交换器

2、生产者使用RoutingKey R 发送消息到交换器

3、如果R 能够匹配 P，则把消息发到该消息队列。

RoutingKey必须由若干个被点.分隔的单词组成。每个单词只能包含字母和数字。其中 \* 匹配一个单词，# 匹配0个或者多个单词。比如 \*.stock.# 匹配 usd.stock 和 eur.stock.db 但是不匹配 stock.nasdaq

**Headers** **交换器**

# RabbitMQ安装

## Win

### 安装erlang环境

otp\_win64\_23.0.exe

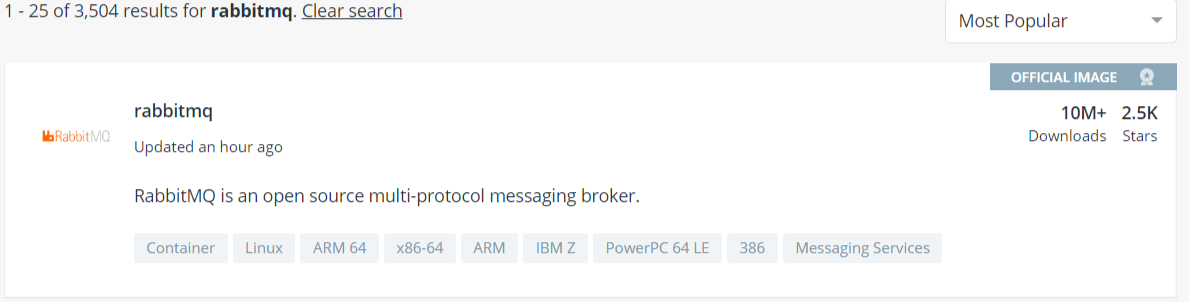
### 安装rabbitmq

rabbitmq-server-3.8.3.exe

### 启动

## Docker

### 下载镜像



拉去镜像：docker pull rabbitmq:3-management (3-management带图形化界面)

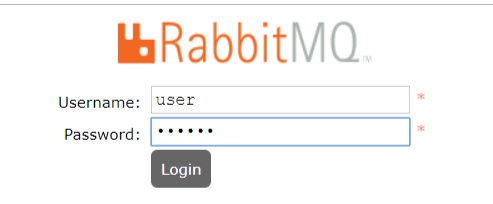
Activemq 里面：图形化8161 数据交互61616

Rabbitmq里面：图形化15671 数据交互5672

运行镜像：docker run -d -p 5672:5672 -p 15672:15672 --name rabbitmq -e RABBITMQ\_DEFAULT\_USER=user -e RABBITMQ\_DEFAULT\_PASS=123456 rabbitmq:3-management

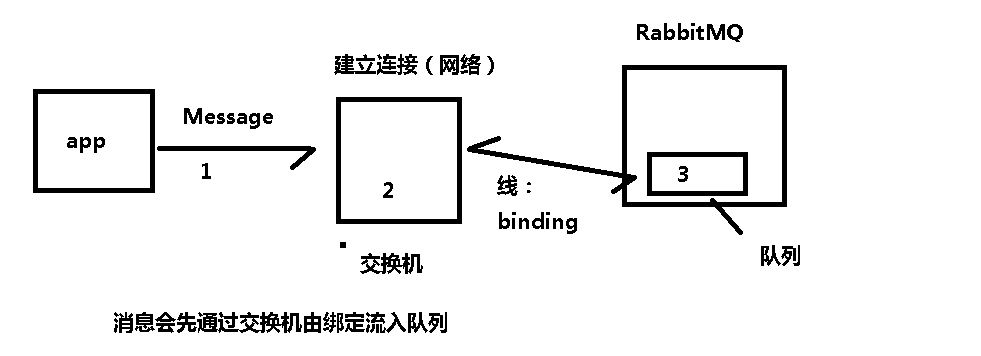
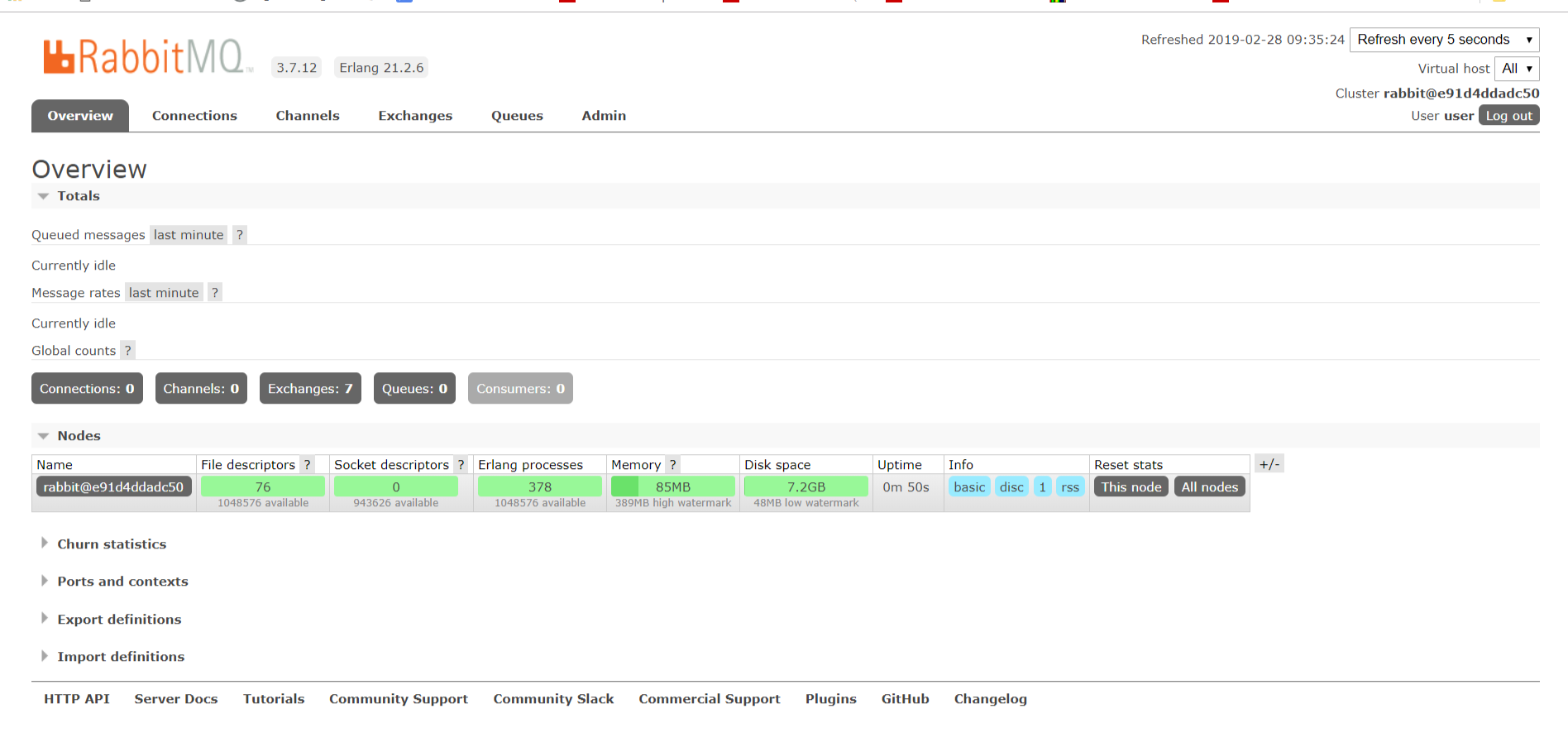
### 登录

图形化界面：<http://192.168.10.136:15672/>



## rabbitmq的名词





## 管理控制台

### 启动

1、Cd: rabbitmq\_server-3.8.3/ sbin 目录下

2、启动rabbitmq-plugins enable rabbitmq\_management

### 访问

Host:http://127.0.0.1:15672/

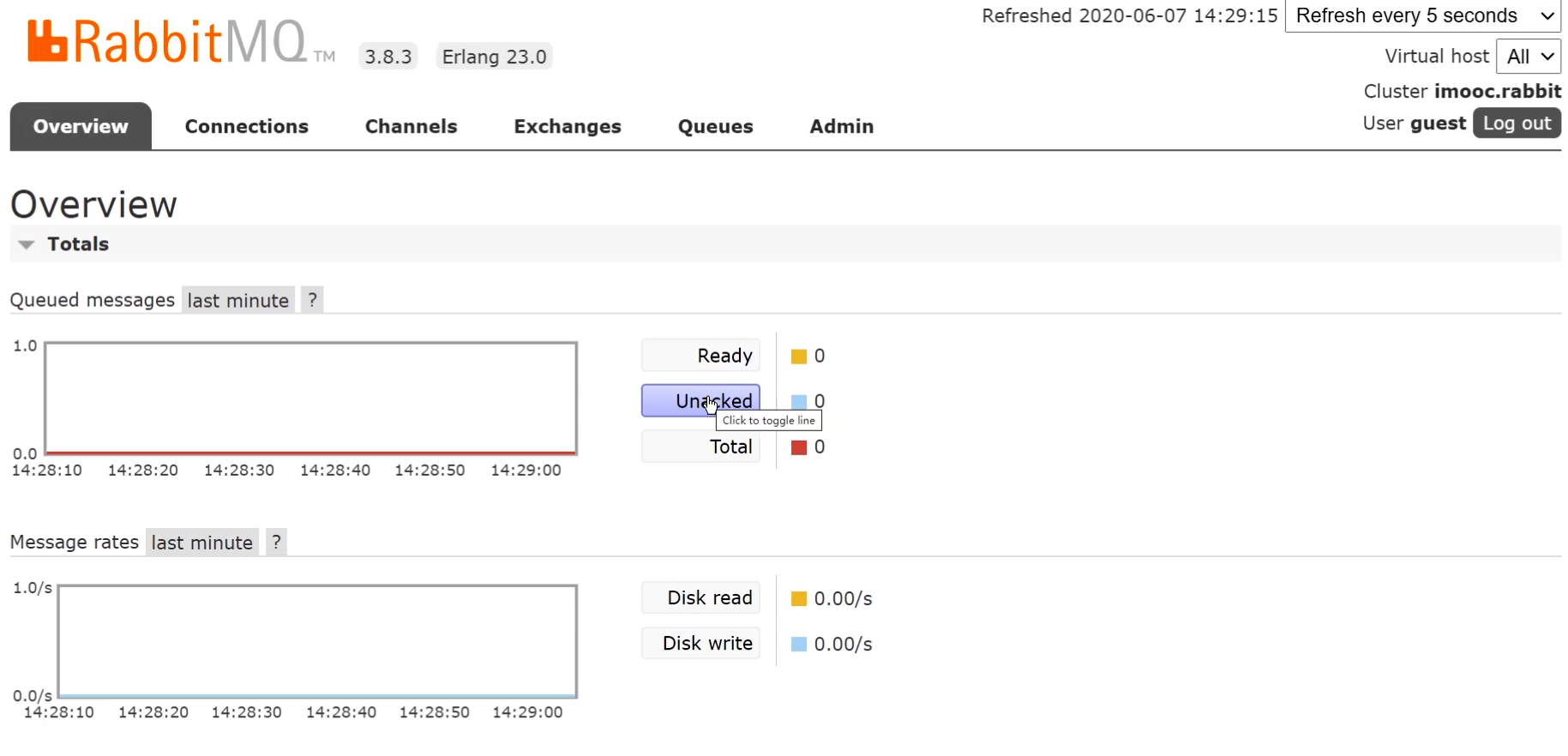
默认账号: guest

默认密码: guest

### 管理页面



### Overview



参数表示正在当前队列里的消息。

Ready:准备好被消费，但是还没有被取走。

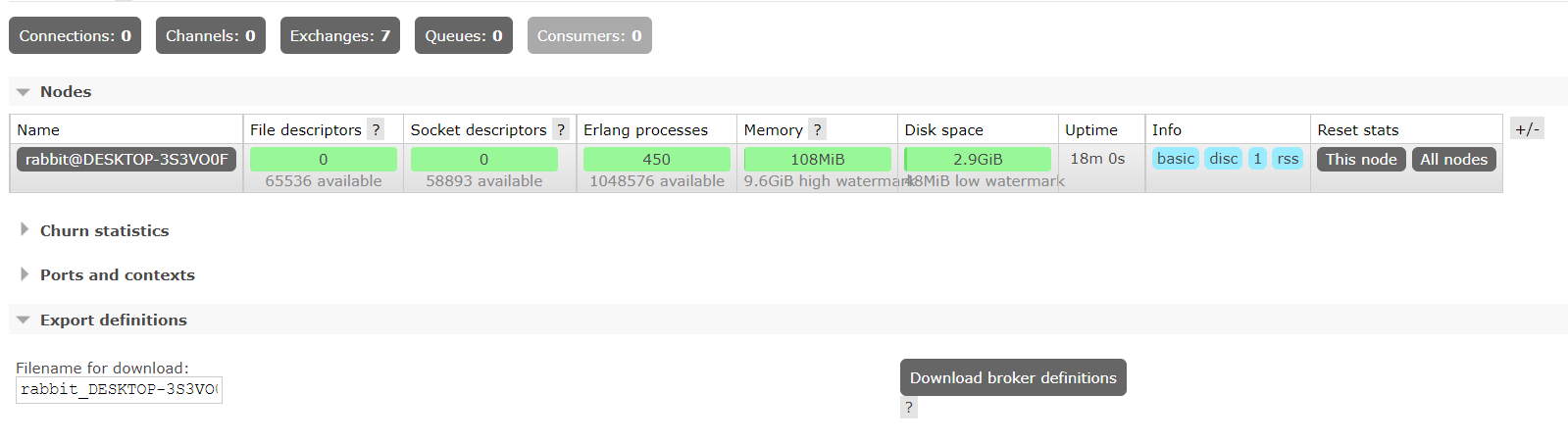
Unacked:被取走了，但是还没有签收。

Total:ready+unacked。

Message Rates:消息处理的速度。

Disk read:磁盘的读。

Disk write:磁盘的写。



Global counts:

Connections:建立的链接

Channels:建立的通道

Exchange:交换机数量

Queues:队列数量

Counsumer:消费这数量

Churn statistics:

Connetion operations:物理tcp链接建立关闭的速率。

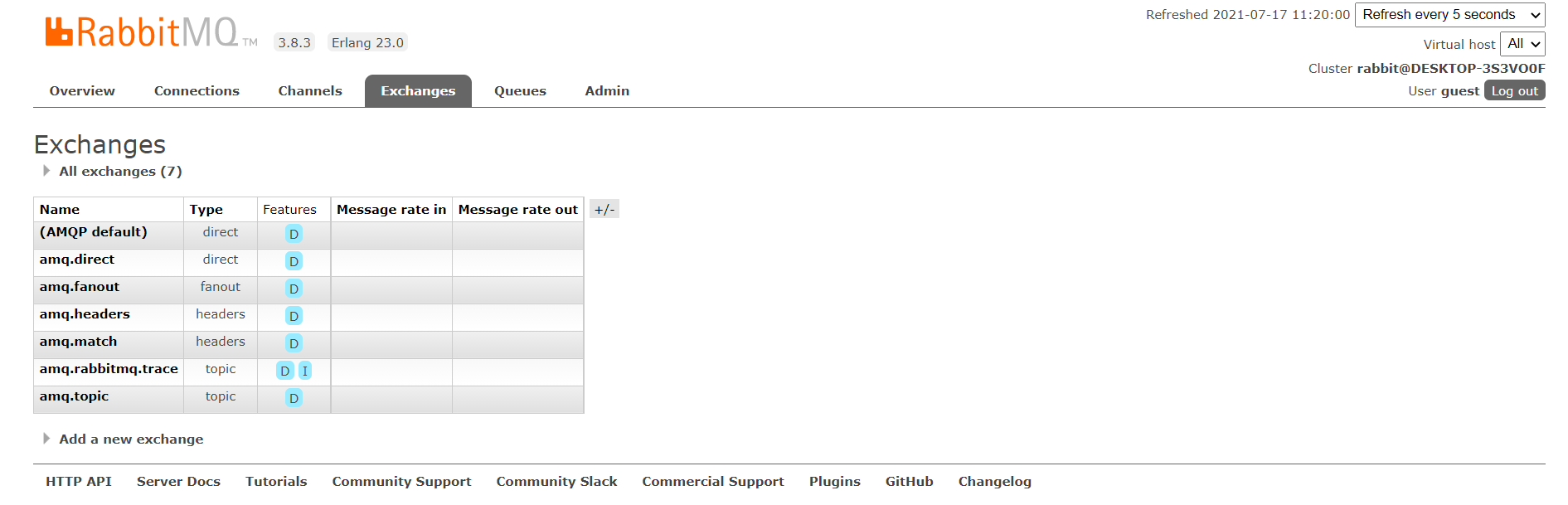
Channe operation:；逻辑连接建立关闭的速率。

Queue operation : 队列建立关闭的速率。

Linstening porst：开放的端口。

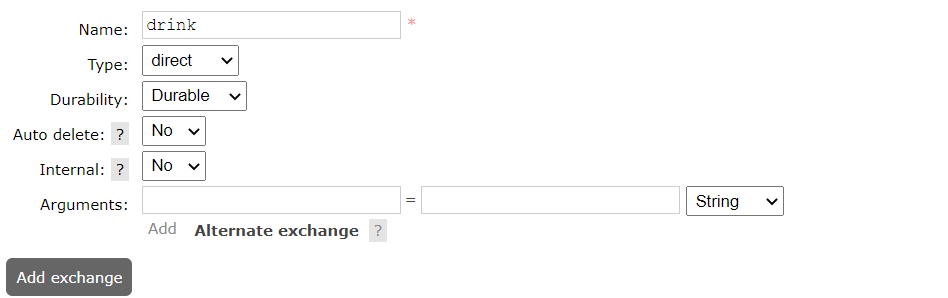
Export definitions:设置的导入和导出，生成一个json文件。

### Exchange



AQMP default:rabbitmq 默认提供的exchange,该exchage的路由key等于queue名称。

### 创建交换机



Name:交换机名称

Type:交换机类型

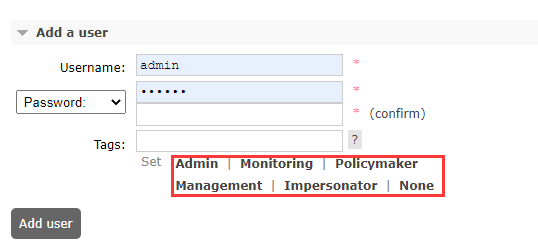
Durablity:是否持久化。

Durablity:持久化到本地。

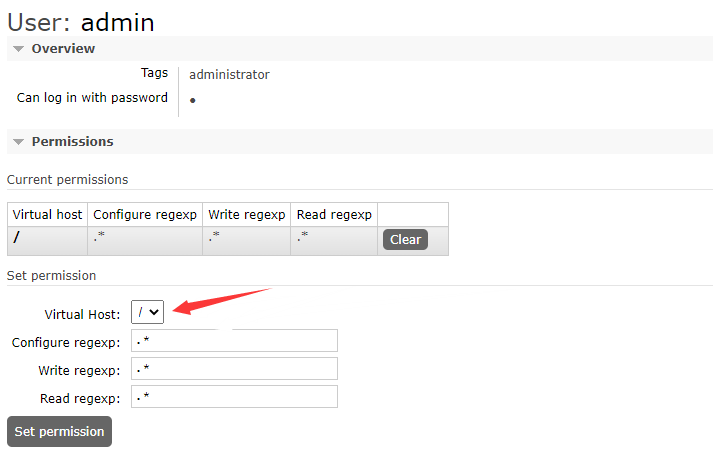
Transient:

Auto delete:如果没有队列绑定该交换机，交换机会删除。

### 添加用户



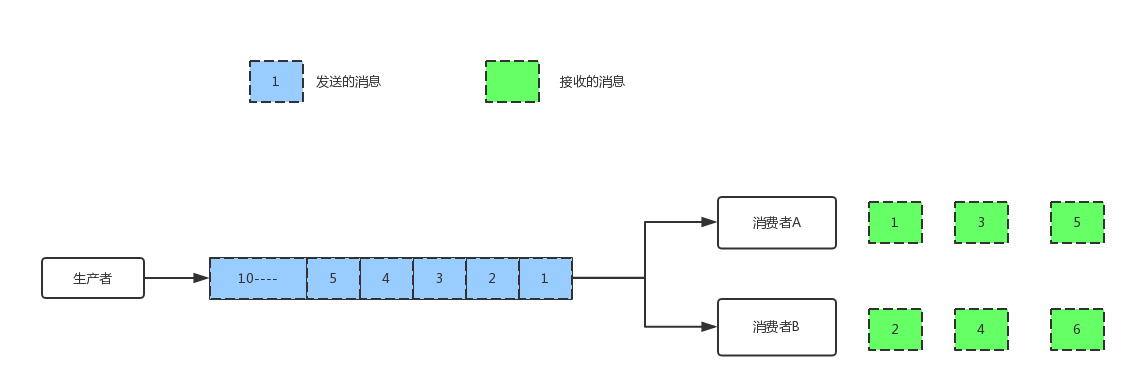
添加用户，设置权限。添加虚拟机。



### 命令

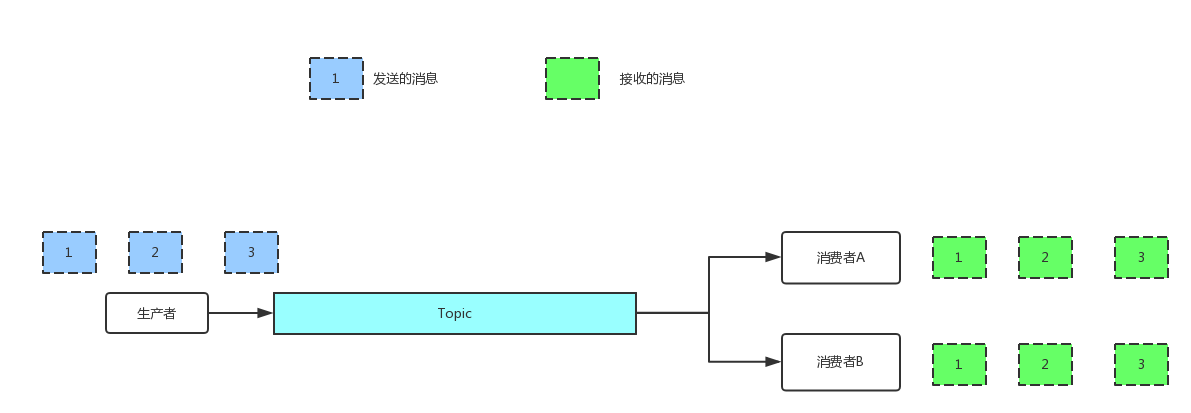
## RabbitMQ

### 队列模式



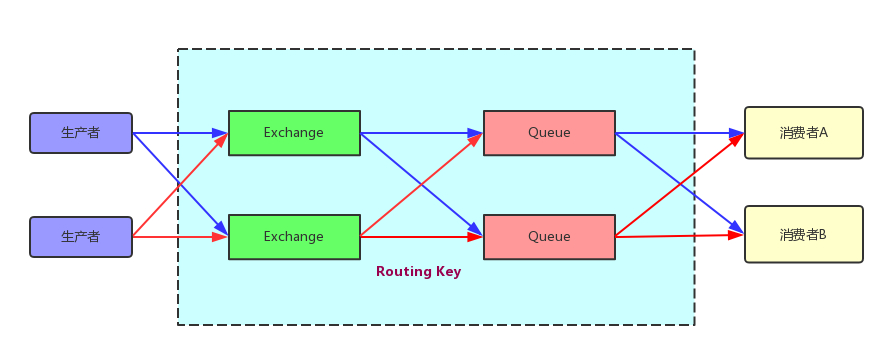
一个发布者发布消息，下面的接收者按队列顺序接收，比如发布了10个消息，两个接收者A,B那就是A,B总共会收到10条消息，不重复

### 主题模式



对于Topic模式，一个发布者发布消息，有两个接收者A,B来订阅，那么发布了10条消息，A,B各收到10条消息。

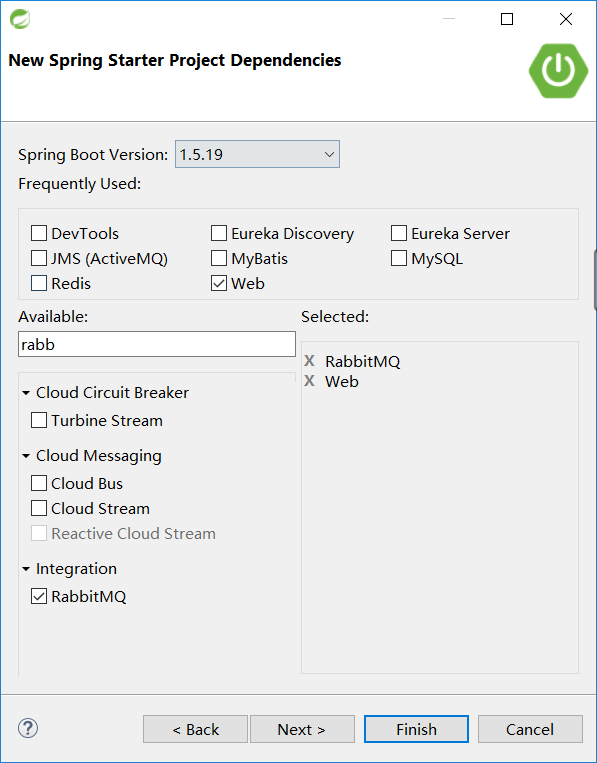
### RabbitMQ的模式



生产者生产消息后不直接直接发到队列中，而是发到一个交换空间：Exchange，Exchange会根据Exchange类型和Routing Key来决定发到哪个队列中，这个讲到发布订阅在详细来看

## RabbitMq的helloworld

### 创建项目



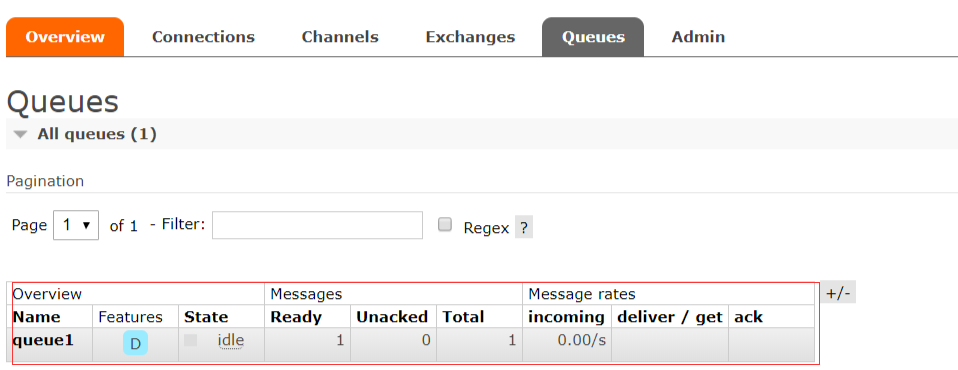
### 配置

|  |
| --- |
| server:  port: 8080  spring:  application:  name: rabbit-demo  rabbitmq:  host: 192.168.10.136  port: 5672  username: user  password: 123456 |

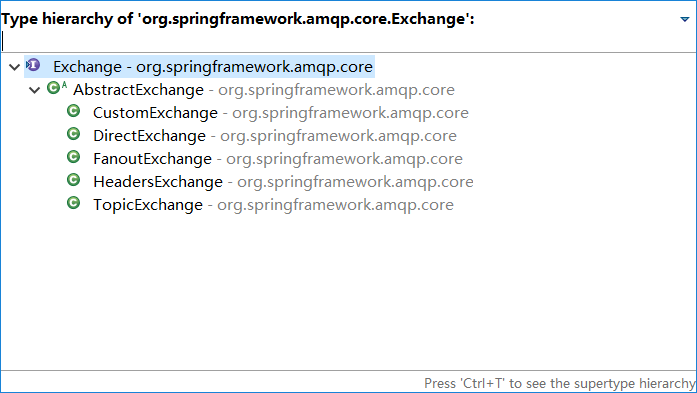
### 发送消息

|  |
| --- |
| @Autowired  **private** RabbitTemplate template;  @Test  **public** **void** contextLoads() {  template.convertAndSend("queue1","value1");  } |

### 结果

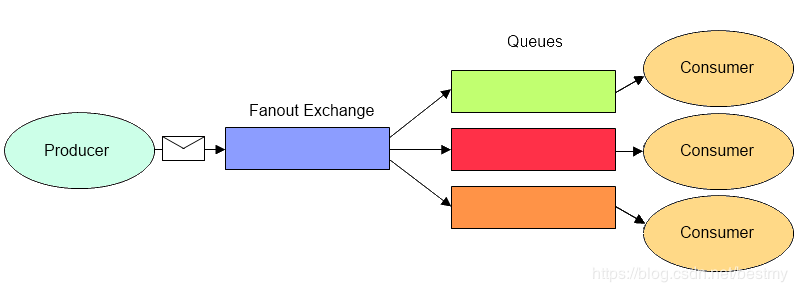


## 交换机



## Fanout （广播模式）

交换器会把所有发送到该交换器的消息路由到所有与该交换器绑定的消息队列中。订阅模式与Binding Key和Routing Key无关，交换器将接受到的消息分发给有绑定关系的所有消息队列队列（不论Binding Key和Routing Key是什么）。类似于子网广播，子网内的每台主机都获得了一份复制的消息。Fanout交换机转发消息是最快的。



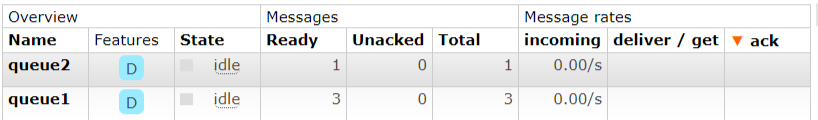
### 代码

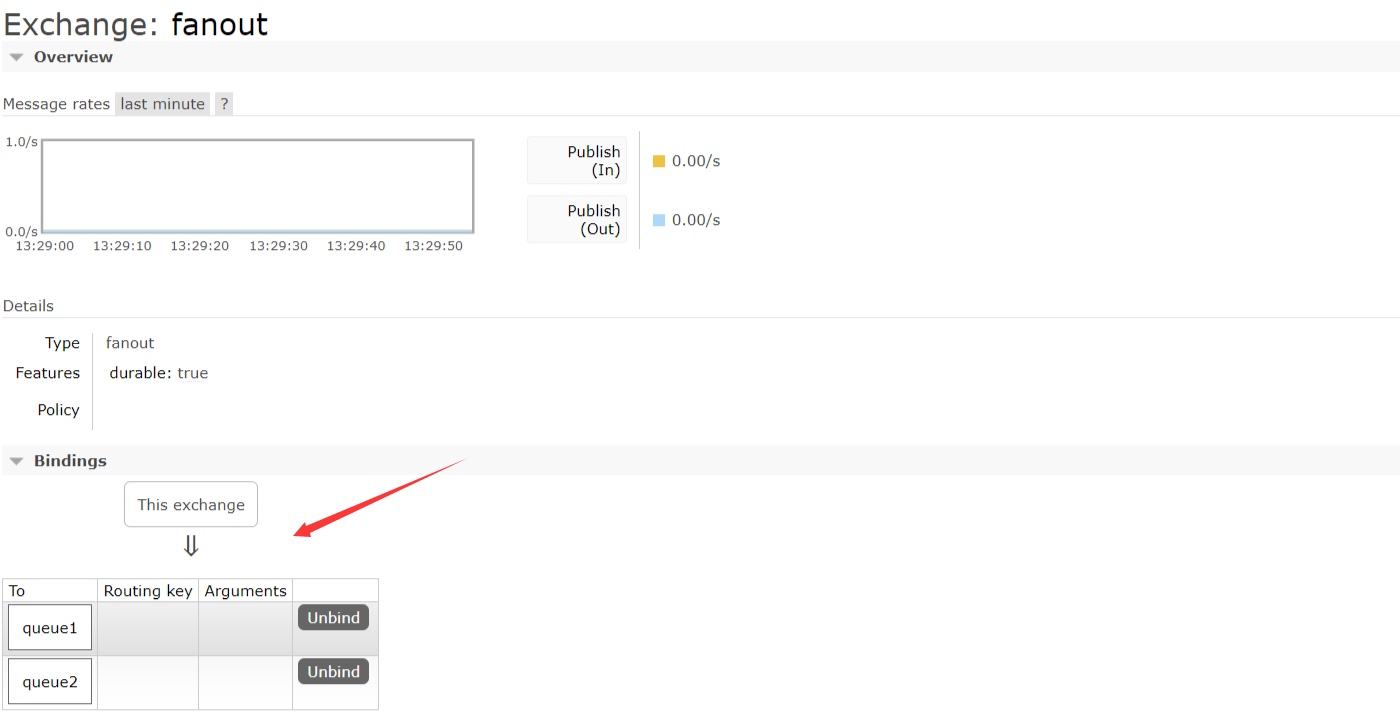
|  |
| --- |
| /\*\*交换机的绑定,队列，绑定  \* **@author** Padingpading  \*  \*/  @Configuration  **public** **class** RabbitMqConf {  @Bean  **public** FanoutExchange fanoutExchange(){  **return** **new** FanoutExchange("fanout");  }    //队列1  @Bean  **public** Queue queue1() {  **return** **new** Queue("queue1");  }  //队列2  @Bean  **public** Queue queue2() {  **return** **new** Queue("queue2");  }    //将交换机fanout和队列queue1绑定  @Bean  **public** Binding binding() {  **return** BindingBuilder.*bind*(queue1()).to(fanoutExchange());  }  //将交换机fanout和队列queue2绑定  @Bean  **public** Binding binding1() {  **return** BindingBuilder.*bind*(queue2()).to(fanoutExchange());  }  } |

### 测试

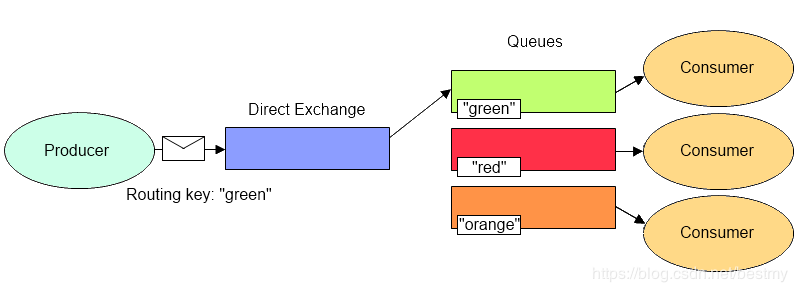
|  |
| --- |
| @Autowired  **private** RabbitTemplate template;  @Test  **public** **void** contextLoads() {  template.convertAndSend("fanout", **null**, "info");  } |

### 结果





## Direct（直连交换机）



精确匹配：当消息的Routing Key与 Exchange和Queue之间的Binding Key完全匹配，如果匹配成功，将消息分发到该Queue。只有当Routing Key和Binding Key完全匹配的时候，消息队列才可以获取消息。Direct是Exchange的默认模式。

RabbitMQ默认提供了一个Exchange，名字是空字符串，类型是Direct，绑定到所有的Queue（每一个Queue和这个无名Exchange之间的Binding Key是Queue的名字）。所以，有时候我们感觉不需要交换器也可以发送和接收消息，但是实际上是使用了RabbitMQ默认提供的Exchange。

匹配规则：交换机+路由key

注：如果两个队列的key相同，两个队列都可以接受到消息。

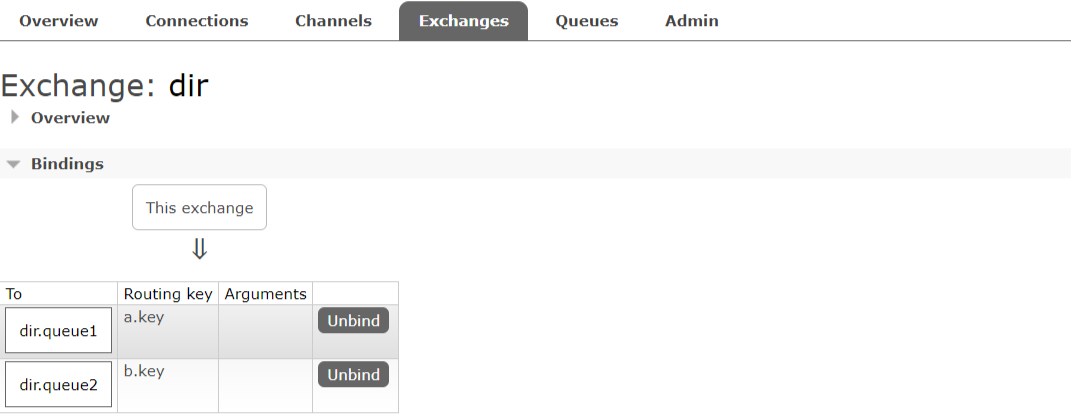
### 配置

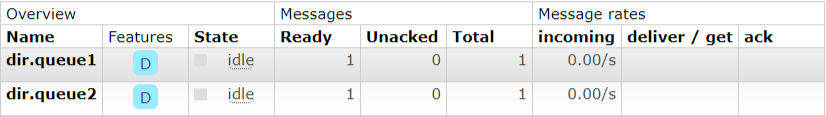
|  |
| --- |
| @Configuration  **public** **class** RabbitMqConf {  //创建交换机  **public** DirectExchange directExchange() {  **return** **new** DirectExchange("direct");  }    //队列1  @Bean  **public** Queue dirqueue1() {  **return** **new** Queue("dir.queue1");  }  //队列2  @Bean  **public** Queue dirqueue2() {  **return** **new** Queue("dir.queue2");  }    //将交换机和队列1绑定,并绑定一个key  @Bean  **public** Binding direBinding1() {  Binding bind = BindingBuilder.*bind*(dirqueue1()).to(directExchange()).with("a.key");  **return** bind;  }    //将交换机和队列2绑定,并绑定一个key  @Bean  **public** Binding direBinding2() {  Binding bind = BindingBuilder.*bind*(dirqueue2()).to(directExchange()).with("b.key");  **return** bind;  } |

### 测试

|  |
| --- |
| @Autowired  **private** RabbitTemplate template;  @Test  **public** **void** contextLoads() {  //携带a.key，将会进入dir.queue1队列  template.convertAndSend("dir", "a.key", "info");  //携带a.key，将会进入dir.queue1队列  template.convertAndSend("dir", "b.key", "info");  } |

### 结果





## Topic （通配符模式）

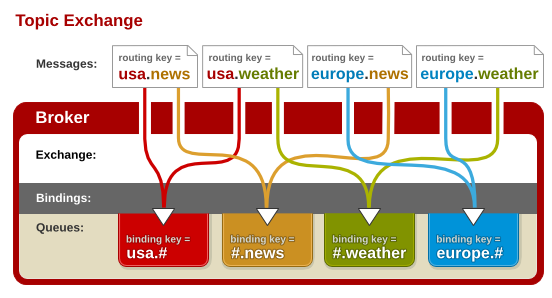
按照正则表达式模糊匹配：用消息的Routing Key与 Exchange和Queue 之间的Binding Key进行模糊匹配，如果匹配成功，将消息分发到该Queue。

Routing Key是一个句点号“. ”分隔的字符串（我们将被句点号“. ”分隔开的每一段独立的字符串称为一个单词）。Binding Key与Routing Key一样也是句点号“. ”分隔的字符串。Binding Key中可以存在两种特殊字符“ \* ”与“#”，用于做模糊匹配，其中“\*”用于匹配一个单词，“#”用于匹配多个单词（可以是零个）。

将路由键和某模式进行匹配。此时队列需要绑定要一个模式上。符号“#”匹配一个或多个词，符号“\*”匹配不多不少一个词。

因此“audit.#”能够匹配到“audit.irs.corporate”，但是“audit.\*” 只会匹配到“audit.irs”

如图：



### config

|  |
| --- |
| //==========================topic=============================//    @Bean  **public** TopicExchange topicExchange() {  **return** **new** TopicExchange("top");  }    //队列1  @Bean  **public** Queue topqueue1() {  **return** **new** Queue("top.queue1");  }  //队列2  @Bean  **public** Queue topqueue2() {  **return** **new** Queue("top.queue2");  }    //将交换机和队列1绑定,绑定一个模糊匹配的key  @Bean  **public** Binding direBinding1() {  Binding bind = BindingBuilder.*bind*(topqueue1()).to(topicExchange()).with("\*.key");  匹配格式  **return** bind;  }    //将交换机和队列2绑定,并绑定一个key  @Bean  **public** Binding direBinding2() {  Binding bind = BindingBuilder.*bind*(topqueue2()).to(topicExchange()).with("#.key");  **return** bind; 匹配格式  } |

### 测试

|  |
| --- |
| **public** **class** RabiitmqDomoApplicationTests {  @Autowired  **private** RabbitTemplate template;  @Test  **public** **void** contextLoads() {  //携带a.key，两个队列都进入 #.key和\*.key，只会匹配一个单词  template.convertAndSend("top", "a.key", "info");  //携带abc.key，只会进入一个队列，#.key，会匹配两个单词  template.convertAndSend("top", "bcd.abc.key", "info");  }  } |

### 结果



top.queue2比top.queue1多一个。

## RabbitMq的消息确认机制

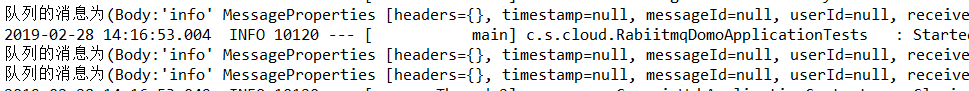
### 监听配置

默认的签收方式时，只要消息能被拿到，不管有没消费成功，都算签收了，签收了，mq服务器会将该消息删除。

消息成功：消息的处理过程中，没有任何异常。

办法：改为手动的签收模式。

|  |
| --- |
| //监听的一个队列  @Component  @RabbitListener(queues={"top.queue1"})  **public** **class** RabbitListenerConfig {    //接收队列的消息，为object，当消息发送到rabbitMq，会自动监听到消息  @RabbitHandler  **public** **void** **handlerMessage**(Object obj) {  System.***out***.println("队列的消息为"+obj);  }  } |



## 发消息确认（100%）

### 发送到交换机确认

确认消息的配置，添加配置publisher-confirms: **true**

|  |
| --- |
| server:  port: 8080  spring:  application:  name: rabbit-demo  rabbitmq:  host: 192.168.10.136  port: 5672  username: user  password: 123456  #消息发送的确认  publisher-confirms: **true** |

确认类：实现ConfirmCallback接口

|  |
| --- |
| **public** **class** MessageToExchangeConfirm **implements** ConfirmCallback {  @Override /  **public** **void** confirm(CorrelationData correlationData, **boolean** ack, String cause) {  //如果交换机得到消息  **if**(ack){  System.***out***.println("消息已经进交换机了");  }**else** {  System.***out***.println("失败原因为"+cause);  }    }  } |

将确认类对象设置到RabbitTemplate模板中

|  |
| --- |
| @Autowired  **private** RabbitTemplate template;  @PostConstruct  **public** **void** confRabbitTemplate() {  template.setConfirmCallback(**new** MessageToExchangeConfirm());  } |

### 消息进入队列的确认

配置publisher-returns: **true**

|  |
| --- |
| server:  port: 8080  spring:  application:  name: rabbit-demo  rabbitmq:  host: 192.168.10.136  port: 5672  username: user  password: 123456  #消息发送到交换ji确认  publisher-confirms: **true**  #消息发送到队列，发送失败才会出发  publisher-returns: **true** |

消息失败的处理类

|  |
| --- |
| **public** **class** MessageToQueueErrorReturn **implements** ReturnCallback{  //消息进入队列失败了，出发方法  @Override  **public** **void** returnedMessage(Message message, **int** replyCode, String replyText, String exchange, String routingKey) {  System.***out***.println("失败的消息为"+message);  System.***out***.println("该消息来自的路由器为"+exchange+"交换机");  System.***out***.println("该消息的路由key为"+routingKey+"交换机");  System.***out***.println("消息的错误编码"+replyCode);  System.***out***.println("消息的错误描述"+replyText);  }  } |

将消息失败的处理类设置给RabbitTemplate

|  |
| --- |
| @Autowired  **private** RabbitTemplate template;    @PostConstruct  **public** **void** confRabbitTemplate() {  template.setConfirmCallback(**new** MessageToExchangeConfirm());  template.setReturnCallback(**new** MessageToQueueErrorReturn());  } |

## 接收消息成功

默认的签收方式时，只要消息能被拿到，不管有没有消费成功，都算签收了，签收了，mq服务器会将该消息删除。改为签收模式。

### 配置

|  |
| --- |
| server:  port: 8080  spring:  application:  name: rabbit-demo  rabbitmq:  host: 192.168.10.136  port: 5672  username: user  password: 123456  #消息发送到交换ji确认  publisher-confirms: **true**  #消息发送到队列，发送失败才会出发  publisher-returns: **true**  listener:  simple:  acknowledge-mode: manual #设置为手动签收消息模式 |

## 消息不能重复操作

# 消息的处理机制

## 简书

### Mq消息解决的问题。

1、发送者->网络->接收者

发送者:

1、自身的网络问题没有发送成功。

2、发送者发送消息之后，因为执行之后出现异常，要回滚。

网络:

1、消息在网络传输过程中造成的丢失。

接收者:

1、接受者可能宕机，或者自身出现问题。

## 发送端的确认(producer)

### 确认机制种类

事务机制:发送消息，如果没有发送成功，抛出异常。

事务确实能够解决producer与broker之间消息确认的问题，只有消息成功被broker接受，事务提交才能成功，否则我们便可以在捕获异常进行事务回滚操作同时进行消息重发，但是使用事务机制的话会降低RabbitMQ的吞吐量，那么有没有更好的方法既能保障producer知道消息已经正确送到，又能基本上不带来性能上的损失呢？从AMQP协议的层面看是没有更好的方法，但是RabbitMQ提供了一个更好的方案，即将channel信道设置成confirm模式。

Confirm机制: 生产者将信道设置成confirm模式，一旦信道进入confirm模式，所有在该信道上面发布的消息都会被指派一个唯一的ID(从1开始)，一旦消息被投递到所有匹配的队列之后，broker就会发送一个确认给生产者（包含消息的唯一ID）,这就使得生产者知道消息已经正确到达目的队列了，如果消息和队列是可持久化的，那么确认消息会将消息写入磁盘之后发出，broker回传给生产者的确认消息中deliver-tag域包含了确认消息的序列号，此外broker也可以设置basic.ack的multiple域，表示到这个序列号之前的所有消息都已经得到了处理。

confirm模式最大的好处在于他是异步的，一旦发布一条消息，生产者应用程序就可以在等信道返回确认的同时继续发送下一条消息，当消息最终得到确认之后，生产者应用便可以通过回调方法来处理该确认消息，如果RabbitMQ因为自身内部错误导致消息丢失，就会发送一条nack消息，生产者应用程序同样可以在回调方法中处理该nack消息。

单条同步确认: 每发送一条消息后，调用waitForConfirms()方法，等待服务器端confirm。实际上是一种串行confirm了。

多条同步确认: 每发送一批消息后，调用waitForConfirms()方法，等待服务器端confirm。

异步确认:提供一个回调方法，服务端confirm了一条或者多条消息后Client端会回调这个方法。

注:

1、已经在transaction事务模式的channel是不能再设置成confirm模式的，即这两种模式是不能共存的。

### 优缺点

事务机制:

优点:消息只有被commit之后,消息才能被rabbmit开始处理，如果发送端在commit之前出现异常，消息会被回滚删除。

缺点:由于开启事务、commit、rollback造成更多的性能消耗，降低了rabbitmq的吞吐量。

confirm模式:

优点：一旦发布一条消息，生产者应用程序就可以在等信道返回确认的同时继续发送下一条消息，当消息最终得到确认之后，生产者应用便可以通过回调方法来处理该确认消息，如果RabbitMQ因为自身内部错误导致消息丢失，就会发送一条nack消息，生产者应用程序同样可以在回调方法中处理该nack消息。

### 事务机制

RabbitMQ中与事务机制有关的方法有三个：

txSelect():用于将当前channel设置成transaction模式。

txCommit():用于提交事务

txRollback():用于回滚事务

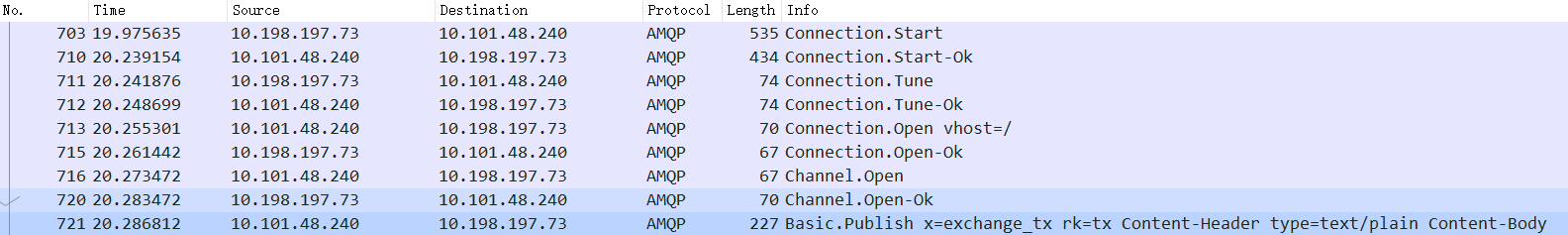
在通过txSelect开启事务之后，我们便可以发布消息给broker代理服务器了，如果txCommit提交成功了，则消息一定到达了broker了，如果在txCommit执行之前broker异常崩溃或者由于其他原因抛出异常，这个时候我们便可以捕获异常通过txRollback回滚事务了。

事务代码

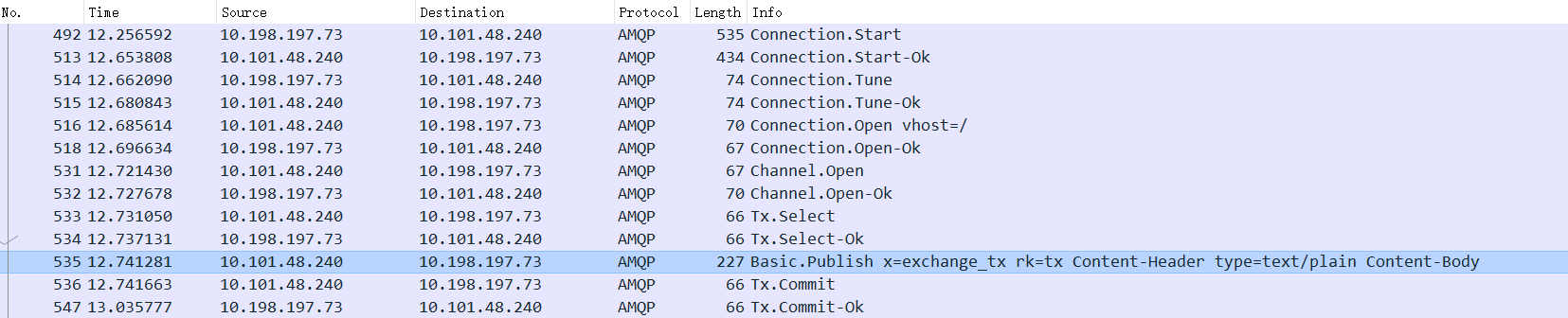
|  |
| --- |
| **try** {  channel.txSelect();  channel.basicPublish(exchange, routingKey, MessageProperties.***PERSISTENT\_TEXT\_PLAIN***, msg.getBytes());  **int** result = 1 / 0;  channel.txCommit(); } **catch** (Exception e) {  e.printStackTrace();  channel.txRollback(); } |

Tcp分析:

没有事务:



开启事务:



事务的多了四个步骤：

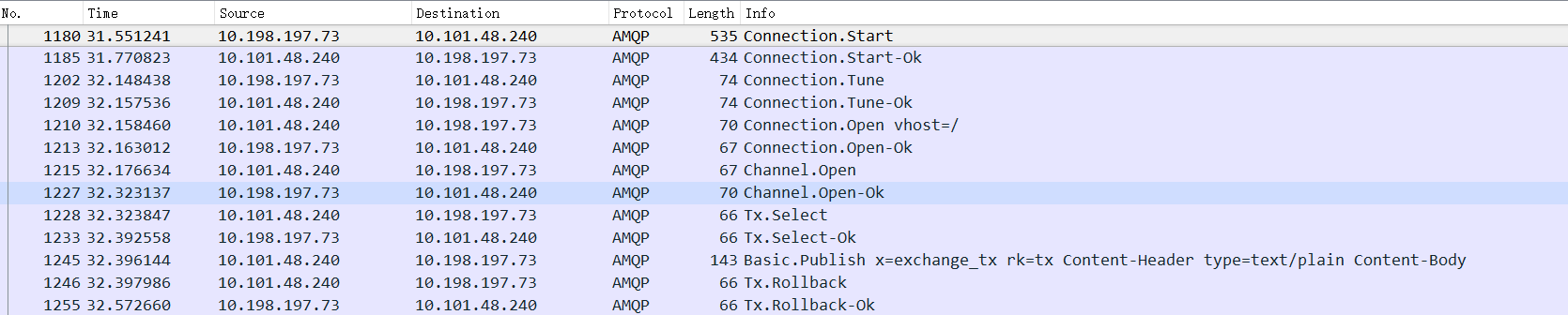
client发送Tx.Select

broker发送Tx.Select-Ok(之后publish)

client发送Tx.Commit

broker发送Tx.Commit-Ok

发生异常:



代码中先是发送了消息至broker中但是这时候发生了异常，之后在捕获异常的过程中进行事务回滚。

### 单条同步确

单挑同步: 普通confirm模式最简单，publish一条消息后，等待服务器端confirm,如果服务端返回false或者超时时间内未返回，客户端进行消息重传。

关键代码如下：

配置方法:

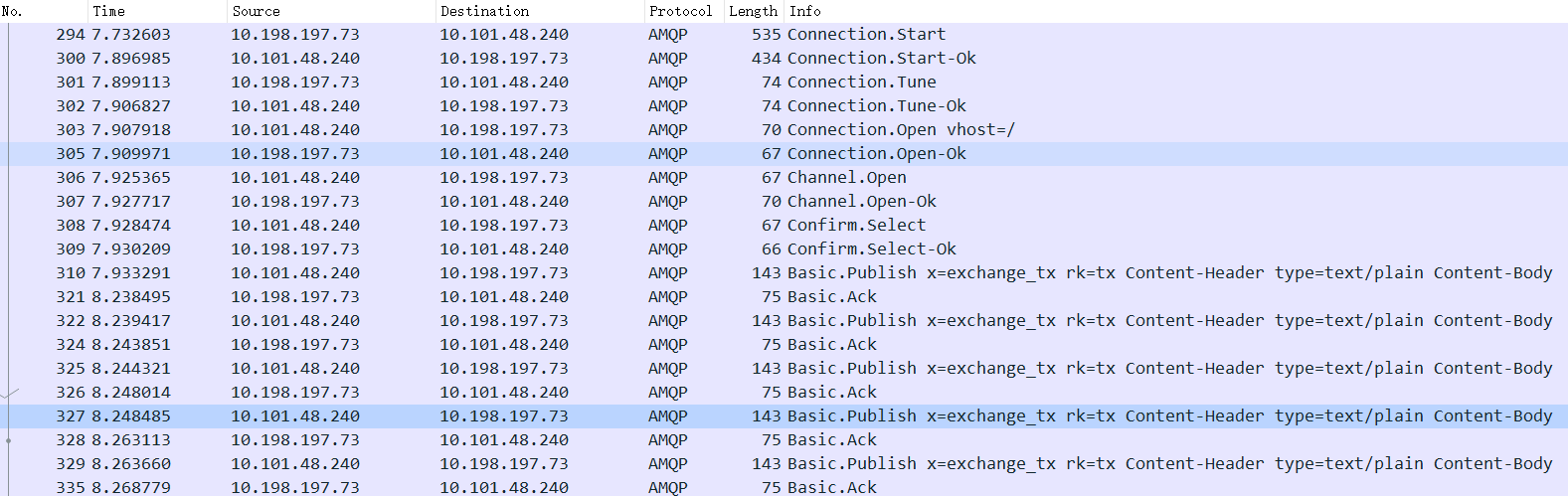
1、配置channel，开启确认模式:channel.confirSelect()。

2、一个channel发送一条消息，立刻调用channel.waitForConfirms()方法,等待确认。

代码

|  |
| --- |
| channel.basicPublish(ConfirmConfig.exchangeName,   ConfirmConfig.routingKey,  MessageProperties.***PERSISTENT\_TEXT\_PLAIN***,  ConfirmConfig.msg\_10B.getBytes()); **if**(!channel.waitForConfirms()){  System.***out***.println(**"send message failed."**); } |

抓包:



(注意这里的Publish与Ack的时间间隔：305ms 4ms 4ms 15ms 5ms… )

### 多条同步确认

多条同步确认:

配置方法:

1、配置channel，开启确认模式:channel.confirSelect()。

2、一个channel发送多条消息，立刻调用channel.waitForConfirms()方法,等待确认。

批量confirm模式稍微复杂一点，客户端程序需要定期（每隔多少秒）或者定量（达到多少条）或者两则结合起来publish消息，然后等待服务器端confirm, 相比普通confirm模式，批量极大提升confirm效率，但是问题在于一旦出现confirm返回false或者超时的情况时，客户端需要将这一批次的消息全部重发，这会带来明显的重复消息数量，并且，当消息经常丢失时，批量confirm性能应该是不升反降的。

关键代码：

|  |
| --- |
| channel.confirmSelect(); **for** (**int** i = 0; i < batchCount; i++) {  channel.basicPublish(ConfirmConfig.exchangeName, ConfirmConfig.routingKey, MessageProperties.PERSISTENT\_TEXT\_PLAIN, ConfirmConfig.msg\_10B.getBytes()); } **if** (!channel.waitForConfirms()) {  System.***out***.println(**"send message failed."**); } |

### 异步确认

异步确认：异步confirm模式的编程实现最复杂，Channel对象提供的ConfirmListener()回调方法只包含deliveryTag（当前Chanel发出的消息序号），我们需要自己为每一个Channel维护一个unconfirm的消息序号集合，每publish一条数据，集合中元素加1，每回调一次handleAck方法，unconfirm集合删掉相应的一条（multiple=false）或多条（multiple=true）记录。从程序运行效率上看，这个unconfirm集合最好采用有序集合SortedSet存储结构。实际上，SDK中的waitForConfirms()方法也是通过SortedSet维护消息序号的。

注:

1、异步确认有可能是单条，也有可能是多条，取决于mq。

配置方法：

1、配置channel，开启确认模式:channel.confirSelect()。

2、在channel上添加监听:addConfirmFListtener,发送消息后,会回调此方法，同时是否发送成功。

deliveryTag:发送的第几条消息。

Multiple:

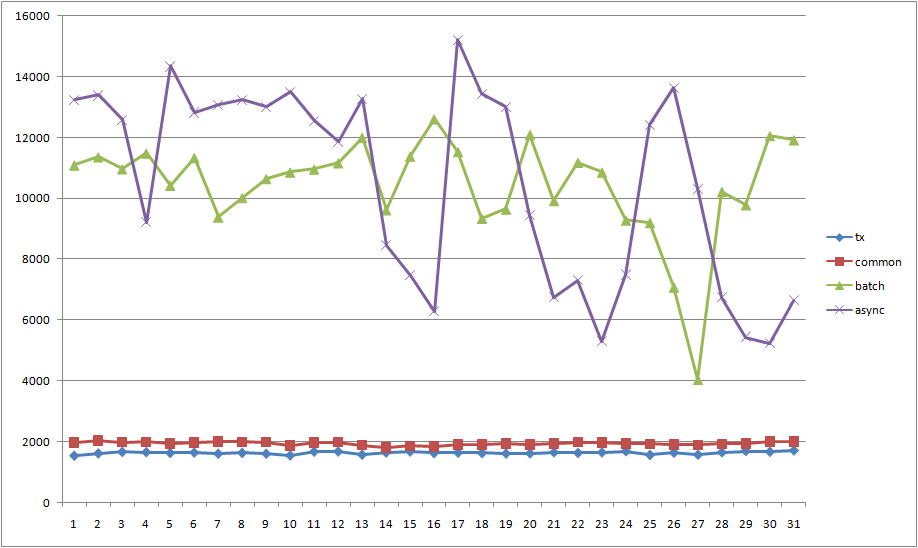
|  |
| --- |
| ConfirmListener confirm = **new** ConfirmListener() {  @Override  **public void** handleAck(**long** deliveryTag, **boolean** multiple) **throws** IOException {  *//收到消息确认* }   @Override  **public void** handleNack(**long** deliveryTag, **boolean** multiple) **throws** IOException {  *//失败消息确认* } }; channel.addConfirmListener(confirm); |

### 性能测试

Client端机器和RabbitMQ机器配置：CPU:24核，2600MHZ, 64G内存，1TB硬盘。

Client端发送消息体大小10B，线程数为1即单线程，消息都持久化处理（deliveryMode:2）。

分别采用事务模式、普通confirm模式，批量confirm模式和异步confirm模式进行producer实验，比对各个模式下的发送性能。



发送平均速率：

事务模式（tx）：1637.484

普通confirm模式(common)：1936.032

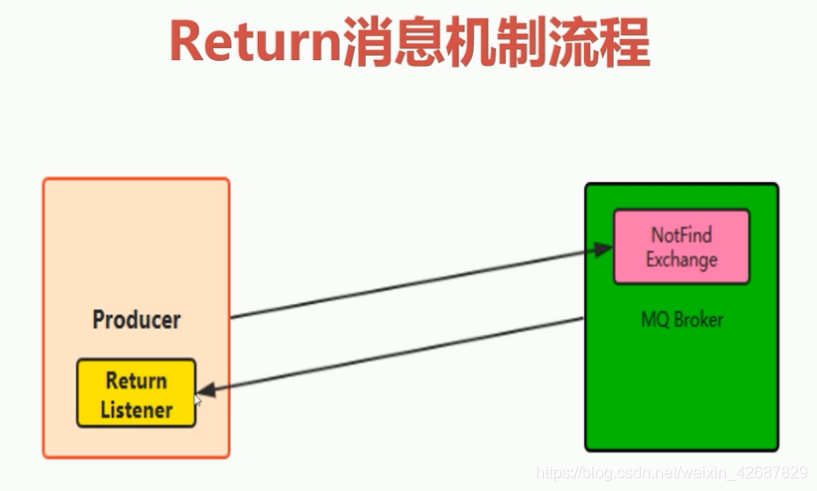
批量confirm模式(batch)：10432.45

异步confirm模式(async)：10542.06

可以看到事务模式性能是最差的，普通confirm模式性能比事务模式稍微好点，但是和批量confirm模式还有异步confirm模式相比，还是小巫见大巫。批量confirm模式的问题在于confirm之后返回false之后进行重发这样会使性能降低，异步confirm模式(async)编程模型较为复杂，至于采用哪种方式，那是仁者见仁智者见智了。

## 消息的返回(rabbitmq)

### 消息的返回



**正常情况**：我们的消息生产者，通过指定一个Exchange和RoutingKey，把消息送达到某一个队列中去，然后我们的消费者监听队列，进行消费处理操作

**异常情况**：在某些情况下，如果我们在发送消息的时候，当前的Exchange不存在或者指定的路由key路由不到，这个时候如果我们需要监听这种不可达的消息，就需要使用Return Listener

### 配置

Mandatory: 当mandatory标志位设置为true时，如果exchange根据自身类型和消息routeKey无法找到一个符合条件的queue，那么会调用basic.return方法将消息返回给生产者（Basic.Return + Content-Header + Content-Body）；当mandatory设置为false时，出现上述情形broker会直接将消息扔掉。

Immediate: 当immediate标志位设置为true时，如果exchange在将消息路由到queue(s)时发现对于的queue上没有消费者，那么这条消息不会放入队列中。当与消息routeKey关联的所有queue（一个或者多个）都没有消费者时，该消息会通过basic.return方法返还给生产者。

概括来说，mandatory标志告诉服务器至少将该消息route到一个队列中，否则将消息返还给生产者；immediate标志告诉服务器如果该消息关联的queue上有消费者，则马上将消息投递给它，如果所有queue都没有消费者，直接把消息返还给生产者，不用将消息入队列等待消费者了。

### Mandatory

在生产者通过channle的basicPublish方法发布消息时，通常有几个参数需要设置，为此我们有必要了解清楚这些参数代表的具体含义及其作用，查看channel接口，会发现存在3个重载的basicPublish方法：

|  |
| --- |
| void basicPublish(String exchange, String routingKey, BasicProperties props, byte[] body) throws IOException;  void basicPublish(String exchange, String routingKey, boolean mandatory, BasicProperties props, byte[] body) throws IOException;  void basicPublish(String exchange, String routingKey, boolean mandatory, boolean immediate, BasicProperties props, byte[] body) throws IOException; |

mandatory和immediate上面已经解释过了，其余的参数分别是：

exchange:交换机名称

routingkey:路由键

props:消息属性字段，比如消息头部信息等等

body:消息主体部分

demo

本节主要讲述mandatory, 下面我们写一个demo，在RabbitMQ broker中有：

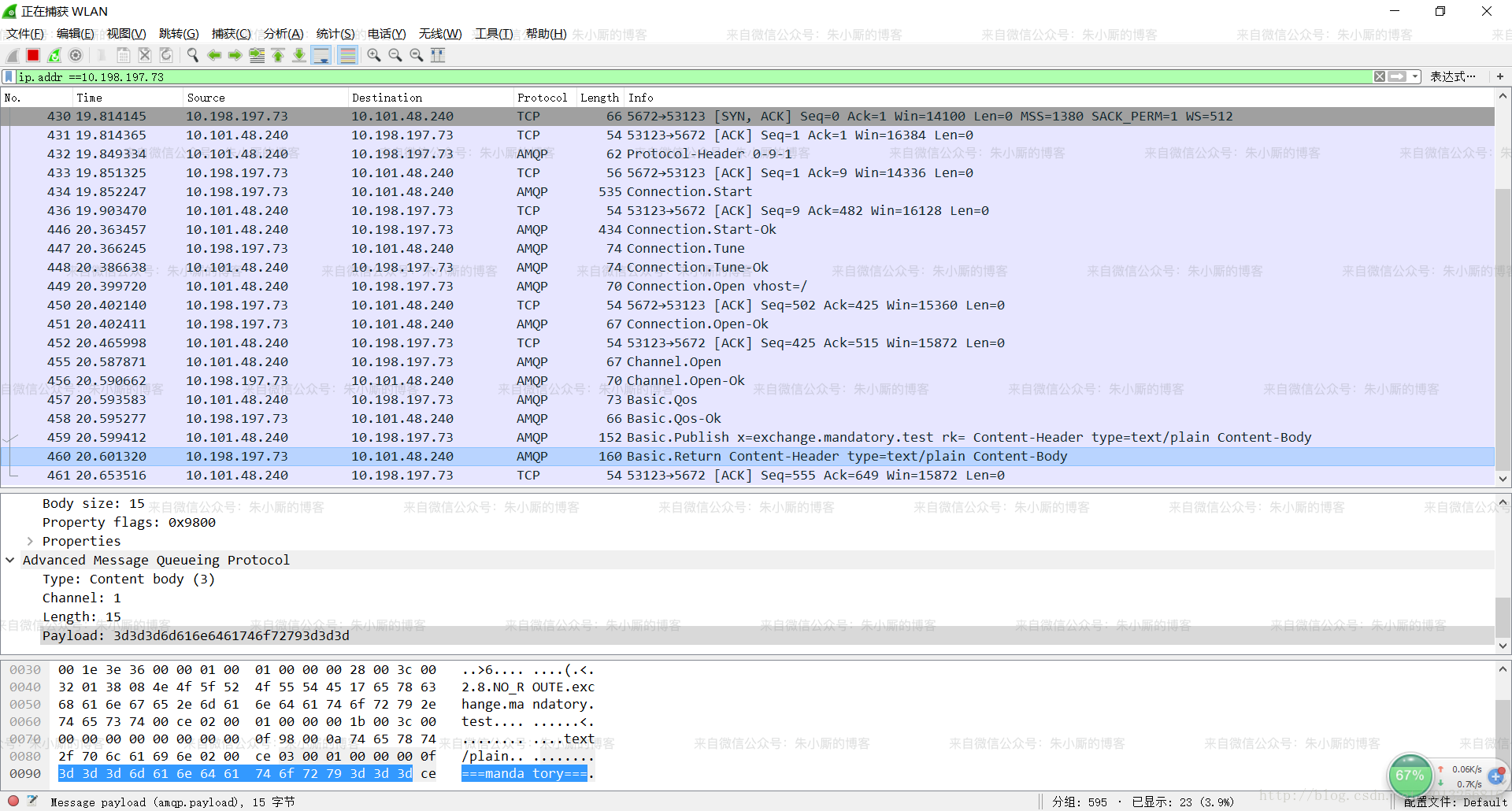
exchange : exchange.mandatory.test

queue: queue.mandatory.test

exchange路由到queue的routingkey是mandatory

|  |
| --- |
| **public class** RBmandatoryTest {  **public static final** String ***ip*** = **"xx.xx.xx.73"**;  **public static final int *port*** = 5672;  **public static final** String ***username*** = **"root"**;  **public static final** String ***password*** = **"root"**;   **public static final** String ***queueName*** = **"queue.mandatory.test"**;  **public static final** String ***exchangeName*** = **"exchange.mandatory.test"**;  **public static final** String ***routingKey*** = **"mandatory"**;  **public static final** Boolean ***mandatory*** = **true**;  **public static final** Boolean ***immediate*** = **false**;   **public static void** main(String[] args) {   **try** {  ConnectionFactory factory = **new** ConnectionFactory();  factory.setHost(***ip***);  factory.setPort(***port***);  factory.setUsername(***username***);  factory.setPassword(***password***);   Connection connection = factory.newConnection();  Channel channel = connection.createChannel();   channel.basicQos(1);  channel.basicPublish(***exchangeName***, **""**, ***mandatory***, ***immediate***, MessageProperties.***PERSISTENT\_TEXT\_PLAIN***, **"===mandatory==="**.getBytes()); *// channel.close(); // connection.close();* } **catch** (IOException e) {  e.printStackTrace();  } **catch** (TimeoutException e) {  e.printStackTrace();  }  } } |

运行，之后通过wireshark抓包工具可以看到如下图所示：

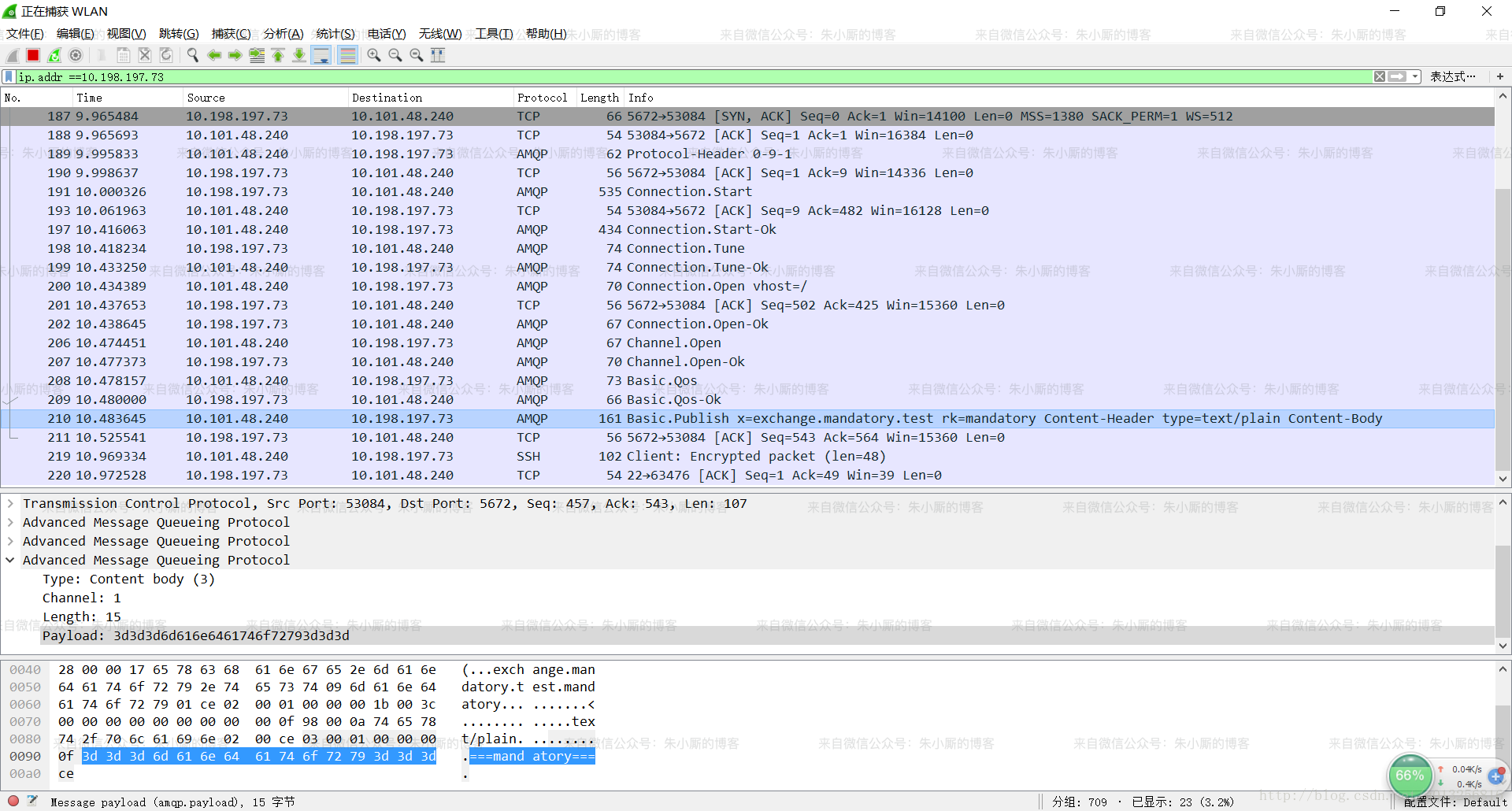


这里可以看到最后执行了basic.return方法，将发布者发出的消息返回给了发布者，查看协议的arguments参数部分可以看到：reply-text字段值为NO\_ROUTE,表示消息并没有路由到合适的队列中；

那么我们该怎么获取到没有被正确路由到合适队列的消息呢？这时候可以通过为channel信道设置ReturnListener监听器来实现，具体代码（main函数部分）：

|  |
| --- |
| **public class** RBmandatoryTest {  **public static final** String ***ip*** = **"xx.xx.xx.73"**;  **public static final int *port*** = 5672;  **public static final** String ***username*** = **"root"**;  **public static final** String ***password*** = **"root"**;   **public static final** String ***queueName*** = **"queue.mandatory.test"**;  **public static final** String ***exchangeName*** = **"exchange.mandatory.test"**;  **public static final** String ***routingKey*** = **"mandatory"**;  **public static final** Boolean ***mandatory*** = **true**;  **public static final** Boolean ***immediate*** = **false**;   **public static void** main(String[] args) {  **try** {  ConnectionFactory factory = **new** ConnectionFactory();  factory.setHost(***ip***);  factory.setPort(***port***);  factory.setUsername(***username***);  factory.setPassword(***password***);   Connection connection = factory.newConnection();  Channel channel = connection.createChannel();   channel.basicQos(1);  channel.basicPublish(***exchangeName***, **""**, ***mandatory***, ***immediate***, MessageProperties.***PERSISTENT\_TEXT\_PLAIN***, **"===mandatory==="**.getBytes());  channel.addReturnListener(**new** ReturnListener() {  **public void** handleReturn(**int** replyCode, String replyText, String exchange, String routingKey, AMQP.BasicProperties basicProperties, **byte**[] body) **throws** IOException {  String message = **new** String(body);  System.***out***.println(**"Basic.return返回的结果是："**+message);  }  });  *// channel.close(); // connection.close();* } **catch** (IOException e) {  e.printStackTrace();  } **catch** (TimeoutException e) {  e.printStackTrace();  }  } } |

重新设置路由key之后，通过wireshark抓包如下：



可以看到并不会有basic.return方法被调用。查看RabbitMQ管理界面发现消息已经到达了队列。

### Immediate

在RabbitMQ3.0以后的版本里，去掉了immediate参数的支持，发送带immediate标记的publish会返回如下错误：

“{amqp\_error,not\_implemented,”immediate=true”,’basic.publish’}”

Removal of “immediate” flag

What changed? We removed support for the rarely-used “immediate” flag on AMQP’s basic.publish.

Why on earth did you do that? Support for “immediate” made many parts of the codebase more complex, particularly around mirrored queues. It also stood in the way of our being able to deliver substantial performance improvements in mirrored queues.

What do I need to do? If you just want to be able to publish messages that will be dropped if they are not consumed immediately, you can publish to a queue with a TTL of 0.

If you also need your publisher to be able to determine that this has happened, you can also use the DLX feature to route such messages to another queue, from which the publisher can consume them.

这段解释的大概意思是：immediate标记会影响镜像队列性能，增加代码复杂性，并建议采用“TTL”和“DLX”等方式替代。

## 接收端确认(consumer)

### 接收端的问题

为了保证消息从队列可靠地到达消费者，RabbitMQ提供消息确认机制(message acknowledgment)。消费者在声明队列时，可以指定noAck参数，当noAck=false时，RabbitMQ会等待消费者显式发回ack信号后才从内存(和磁盘，如果是持久化消息的话)中移去消息。否则，RabbitMQ会在队列中消息被消费后立即删除它

采用消息确认机制后，只要令noAck=false，消费者就有足够的时间处理消息(任务)，不用担心处理消息过程中消费者进程挂掉后消息丢失的问题，因为RabbitMQ会一直持有消息直到消费者显式调用basicAck为止。

当noAck=false时，对于RabbitMQ服务器端而言，队列中的消息分成了两部分：

一部分是等待投递给消费者的消息；

一部分是已经投递给消费者，但是还没有收到消费者ack信号的消息。

如果服务器端一直没有收到消费者的ack信号，并且消费此消息的消费者已经断开连接，则服务器端会安排该消息重新进入队列，等待投递给下一个消费者（也可能还是原来的那个消费者）。

RabbitMQ不会为未ack的消息设置超时时间，它判断此消息是否需要重新投递给消费者的唯一依据是消费该消息的消费者连接是否已经断开。这么设计的原因是RabbitMQ允许消费者消费一条消息的时间可以很久很久。

RabbitMQ管理平台界面上可以看到当前队列中Ready状态和Unacknowledged状态的消息数，分别对应上文中的等待投递给消费者的消息数和已经投递给消费者但是未收到ack信号的消息数。也可以通过命令行来查看上述信息：

### 配置

代码示例（关闭自动消息确认，进行手动ack）：

|  |
| --- |
| QueueingConsumer consumer = **new** QueueingConsumer(channel); channel.basicConsume(ConfirmConfig.queueName, **false**, consumer);  **while**(**true**){  QueueingConsumer.Delivery delivery = consumer.nextDelivery();  String msg = **new** String(delivery.getBody()); *// do something with msg.* channel.basicAck(delivery.getEnvelope().getDeliveryTag(), **false**); } |

broker将在下面的情况中对消息进行confirm：

broker发现当前消息无法被路由到指定的queues中（如果设置了mandatory属性，则broker会发送basic.return）

非持久属性的消息到达了其所应该到达的所有queue中（和镜像queue中）

持久消息到达了其所应该到达的所有queue中（和镜像中），并被持久化到了磁盘（fsync）

持久消息从其所在的所有queue中被consume了（如果必要则会被ack）

basicRecover：是路由不成功的消息可以使用recovery重新发送到队列中。

basicReject：是接收端告诉服务器这个消息我拒绝接收,不处理,可以设置是否放回到队列中还是丢掉，而且只能一次拒绝一个消息,官网中有明确说明不能批量拒绝消息，为解决批量拒绝消息才有了basicNack。

basicNack：可以一次拒绝N条消息，客户端可以设置basicNack方法的multiple参数为true，服务器会拒绝指定了delivery\_tag的所有未确认的消息(tag是一个64位的long值，最大值是9223372036854775807)。

## 接收端的限流

### 场景

业务高峰期,有个微服务崩溃了,崩溃期间队列挤压了大量消息,微服务上线后,收到大量并发消息

将同样多的消息推给能力不同的副本,会导致部分副本异常

### Qos

## 消息的过期

### 概述

RabbitMQ可以对消息和队列设置TTL. 目前有两种方法可以设置。

第一种方法是通过队列属性设置，队列中所有消息都有相同的过期时间。

第二种方法是对消息进行单独设置，每条消息TTL可以不同。

如果上述两种方法同时使用，则消息的过期时间以两者之间TTL较小的那个数值为准。消息在队列的生存时间一旦超过设置的TTL值，就称为dead message， 消费者将无法再收到该消息。

### 对比

对于第一种设置队列TTL属性的方法，一旦消息过期，就会从队列中抹去，而第二种方法里，即使消息过期，也不会马上从队列中抹去，因为每条消息是否过期时在即将投递到消费者之前判定的，为什么两者得处理方法不一致？因为第一种方法里，队列中已过期的消息肯定在队列头部，RabbitMQ只要定期从队头开始扫描是否有过期消息即可，而第二种方法里，每条消息的过期时间不同，如果要删除所有过期消息，势必要扫描整个队列，所以不如等到此消息即将被消费时再判定是否过期，如果过期，再进行删除。

### 如何设置自己的ttl

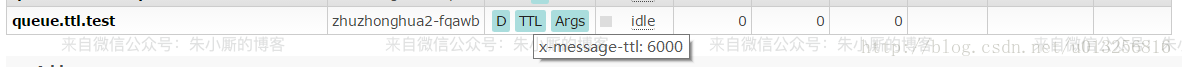
1、TTL应该明显长于服务的平均重启时间，如果ttl<服务重启时间,服务重启之后就会丢失。

2、建议TTL长于业务高峰期时间。例如秒杀业务中，设置ttl时间为半个小时以上，避免在秒杀中处理不过来。

### 设置队列属性

通过队列属性设置消息TTL的方法是在queue.declare方法中加入x-message-ttl参数，单位为ms.

通过RabbitMQ的管理页面可以看到有新的queue生成，并标记为TTL（上面的代码同时会将此queue设置为durable=true,以及包含相关参数，比如vhost=/），如下图所示：



另外也可以同rabbitmq的命令行模式来设置：

rabbitmqctl set\_policy TTL ".\*" '{"message-ttl":60000}' --apply-to queues

$ curl -i -u guest:guest -H "content-type:application/json" -XPUT -d'{"auto\_delete":false,"durable":true,"arguments":{"x-message-ttl": 60000}}'

[http://localhost:15672/api/queues/{vhost}/{queuename}](http://localhost:15672/api/queues/%7bvhost%7d/%7bqueuename%7d)

如果不设置TTL,则表示此消息不会过期。如果将TTL设置为0，则表示除非此时可以直接将消息投递到消费者，否则该消息会被立即丢弃，这个特性可以部分替代RabbitMQ3.0以前支持的immediate参数，之所以所部分代替，是应为immediate参数在投递失败会有basic.return方法将消息体返回（这个功能可以利用死信队列来实现）。

### 设置消息属性

针对每条消息设置TTL的方法是在basic.publish方法中加入expiration的属性参数，单位为ms.

关键代码：

|  |
| --- |
| AMQP.BasicProperties.Builder builder = **new** AMQP.BasicProperties.Builder(); builder.deliveryMode(2); builder.expiration(**"6000"**); AMQP.BasicProperties properties = builder.build();  channel.basicPublish(**exchangeName**,routingKey,mandatory,properties,**"ttlTestMessage"**.getBytes()); |

## 死信队列

### 死信队列

DLX, Dead-Letter-Exchange。利用DLX, 当消息在一个队列中变成死信（dead message）之后，它能被重新publish到另一个Exchange，这个Exchange就是DLX。消息变成死信一向有一下几种情况：

消息被拒绝（basic.reject/ basic.nack）并且requeue=false

消息TTL过期（参考：RabbitMQ之TTL（Time-To-Live 过期时间））

队列达到最大长度

DLX也是一个正常的Exchange，和一般的Exchange没有区别，它能在任何的队列上被指定，实际上就是设置某个队列的属性，当这个队列中有死信时，RabbitMQ就会自动的将这个消息重新发布到设置的Exchange上去，进而被路由到另一个队列，可以监听这个队列中消息做相应的处理，这个特性可以弥补RabbitMQ 3.0以前支持的immediate参数（可以参考RabbitMQ之mandatory和immediate）的功能。

### 配置

核心代码实现：通过在queueDeclare方法中加入“x-dead-letter-exchange”实现。

|  |
| --- |
| channel.exchangeDeclare("some.exchange.name", "direct");  Map<String, Object> args = new HashMap<String, Object>();  args.put("x-dead-letter-exchange", "some.exchange.name");  channel.queueDeclare("myqueue", false, false, false, args); |

你也可以为这个DLX指定routing key，如果没有特殊指定，则使用原队列的routing key

args.put("x-dead-letter-routing-key", "some-routing-key");

命令配置: rabbitmqctl set\_policy DLX ".\*" '{"dead-letter-exchange":"my-dlx"}' --apply-to queues

修改RabbitMQ之TTL（Time-To-Live 过期时间）中的例子：

|  |
| --- |
| **public static void** createQueue(){  **try** {  ConnectionFactory factory = **new** ConnectionFactory();  factory.setHost(ip);  factory.setPort(port);  factory.setUsername(username);  factory.setPassword(password);   Connection connection = factory.newConnection();  Channel channel = connection.createChannel();   Map<String, Object> argss = **new** HashMap<String, Object>();  argss.put(**"vhost"**, **"/"**);  argss.put(**"username"**,**"root"**);  argss.put(**"password"**, **"root"**);  argss.put(**"x-message-ttl"**,6000);  argss.put(**"x-dead-letter-exchange"**,**"exchange.dlx.test"**);  argss.put(**"x-dead-letter-routing-key"**,**"queue.dlx.test"**);  channel.queueDeclare(**"queue.dlx.test"**, durable, exclusive, autoDelete, argss);  } **catch** (IOException e) {  e.printStackTrace();  } **catch** (TimeoutException e) {  e.printStackTrace();  } } |

通过RabbitMQ的管理界面可以看到：

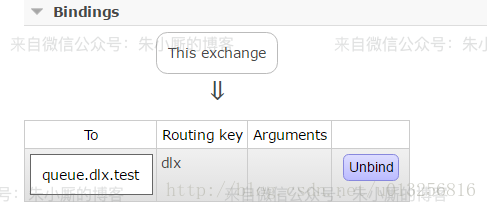


queue.dlx.test这个queue中有个“DLX”和“DLK”的标记. DLX关联的是exchangeName, DLK关联的是routingKey.

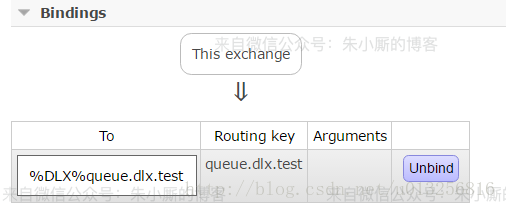
### 详细说明

在RabbitMQ中有两个exchange: exchange.dlx.self和exchange.dlx.test，两个queue：queue.dlx.test和%DLX%queue.dlx.test

exchange.dlx.self是正常情况下，生产者发送消息到此exchange中，绑定关系如图：



exchang.dlx.test是产生死信之后，原queue[queue.dlx.test]的死信发送到此exchange中，绑定关系如图：



数据首先发送到 exchange[exchange.dlx.self]，根据routingkey[dlx]路由到queue.dlx.test,如果正常情况下，消费者可以消费queue.dlx.test的内容。但是如果queue.dlx.test中有消息变成了dead message即死信了，那么这个死信则会通过exchangeName=exchange.dlx.test, routingKey=”queue.dlx.test”路由到死信队列%DLX%queue.dlx.test中，如果要消费这个dead message, 此时消费者必须消费%DLX%queue.dlx.test中的内容而不是queue.dlx.test中的内容。

如果不指定x-dead-letter-routing-key参数，则使用原来的routingkey

# Springboot集成

# 负载均衡

## 负载均衡

面对大量业务访问、高并发请求可以使用高性能的服务器来提升RabbitMQ服务的负载能力。当单机容量达到极限时，我们可以采取集群的策略来对负载能力做进一步的提升，但是这里还存在一个负载不均衡的问题。试想如果一个集群中有3个节点，那么所有的客户端都与其中的单个节点node1建立TCP连接，那么node1的网络负载必然会大大增加而显得难以承受，其他节点又由于没有那么多的负载而造成硬件资源的浪费，所以负载均衡显得尤为重要。

负载均衡（Load balance）是一种计算机网络技术，用于在多个计算机（计算机集群）、网络连接、CPU、磁盘驱动器或其他资源中分配负载，以达到最佳资源使用、最大化吞吐率、最小响应时间以及避免过载的目的。使用带有负载均衡的多个服务器组件，取代单一的组件，可以通过冗余提高可靠性。负载均衡通常分为软件负载均衡和硬件负载均衡两种。

软件负载均衡是指在一个或者多个交互的网络系统中的多台放服务器上安装一个或多个相应的负载均衡软件来实现一种均衡负载技术。软件可以很方便的安装在服务器上，并且实现一定的均衡负载功能。软件负载均衡技术配置简单、操作也仿版，最重要的是成本很低。

硬件负载均衡是指在多台服务器间安装相应的负载均衡设备，也就是负载均衡器（如F5）来完成均衡负载技术，与软件负载均衡技术相比，能达到更好的负载均衡效果。由于硬件负载均衡技术需要额外的增加负载均衡器，成本比较高，所以适用于流量高的大型网站系统。

这里主要讨论的是如何有效的对RabbitMQ集群使用软件负载均衡技术，目前主流的方式有在客户端内部实现负载均衡，或者使用HAProxy、LVS等负载均衡软件来实现。

## 负载均衡算法

对于RabbitMQ而言可以在客户端连接时简单的使用负载均衡算法来实现负载均衡。负载均衡算法有很多种，主流的有：

轮询法

### 轮询法

将请求按顺序轮流地分配到后端服务器上，它均衡地对待后端的每一台服务器，而不关系服务器实际的连接数和当前的系统负载。

示例如代码清单所示，如果多个客户端需要连接到这个有3个节点的RabbitMQ集群，可以调用RoundRobin.getConnectionAddress()来获取相应的连接地址。

|  |
| --- |
| public class RoundRobin {  private static List<String> list = new ArrayList<String>(){{  add("192.168.0.2");  add("192.168.0.3");  add("192.168.0.4");  }};  private static int pos = 0;  private static final Object lock = new Object();  public static String getConnectionAddress(){  String ip = null;  synchronized (lock) {  ip = list.get(pos);  if (++pos >= list.size()) {  pos = 0;  }  }  return ip;  }  } |

### 随机法

通过随机算法，根据后端服务器的列表大小值来随机选取其中的一台服务器进行访问。由概率统计理论可以得知，随着客户端调用服务端的次数增多，其实际效果越来越接近于平均分配调用量到后端的每一台服务器，也就是轮询的结果。对应的示例代码如下：

|  |
| --- |
| public class RandomAccess {  private static List<String> list = new ArrayList<String>(){{  add("192.168.0.2");  add("192.168.0.3");  add("192.168.0.4");  }};  public static String getConnectionAddress(){  Random random = new Random();  int pos = random.nextInt(list.size());  return list.get(pos);  }  } |

### 源地址哈希法

源地址哈希的思想是根据获取的客户端IP地址，通过哈希函数计算得到的一个数值，用该数值对服务器列表的大小进行取模运算，得到的结果便是客户端要访问服务器的序号。采用源地址哈希法进行负载均衡，同一IP地址的客户端，当后端服务器列表不变时，它每次都会映射到同一台后端服务器进行访问。

|  |
| --- |
| public class IpHash {  private static List<String> list = new ArrayList<String>(){{  add("192.168.0.2"); add("192.168.0.3"); add("192.168.0.4");  }};  public static String getConnectionAddress() throws UnknownHostException {  int ipHashCode = InetAddress.getLocalHost().getHostAddress().hashCode();  int pos = ipHashCode % list.size();  return list.get(pos);  }  } |

### 加权轮询法

不同的后端服务器可能机器的配置和当前系统的负载并不相同，因此它们的抗压能力也不相同。给配置高、负载低的机器配置更高的权重，让其处理更多的请求；而配置低、负载高的集群，给其分配较低的权重，降低其系统负载，加权轮询能很好地处理这一问题，并将请求顺序且按照权重分配到后端。

### 加权随机法

与加权轮询法一样，加权随机法也根据后端机器的配置、系统的负载分配不同权重。不同的是，它是按照权重随机请求后端服务器，而非顺序。

### 最小连接数法

最小连接数算法比较灵活和智能，由于后端服务器的配置不尽相同，对于请求的处理有块有慢，它是根据后端服务器当前的连接情况，动态地选取其中当前积压连接数最少的一台服务器来处理当前的请求，尽可能地提高后端服务的利用效率，将负载合理地分流到每一台服务器。

## HAProxy

### 1 HAProxy

HAProxy提供高可用性、负载均衡以及基于TCP和HTTP应用的代理，支持虚拟主机，它是免费、快速并且可靠的一种解决方案,包括Twitter，Reddit，StackOverflow，GitHub在内的多家知名互联网公司在使用。HAProxy实现了一种事件驱动、单一进程模型，此模型支持非常大的并发连接数。

### 安装HAProxy

首先需要去HAProxy的官网下载HAProxy的安装文件，目前最新的版本为：haproxy-1.7.8.tar.gz。下载地址为http://www.haproxy.org/#down，相关文档地址为http://www.haproxy.org/#doc1.7。

将haproxy-1.7.8.tar.gz拷贝至/opt目录下，与RabbitMQ存放在同一个目录中。之后解压缩：

[root@node1 opt]# tar zxvf haproxy-1.7.8.tar.gz

源码解压之后，需要运行make来将HAProxy编译为可执行程序。在执行make之前需要先选择目标平台，通常对于UNIX系的操作系统可以选择TARGET=generic。下面是详细操作：

|  |
| --- |
| [root@node1 opt]# cd haproxy-1.7.8  [root@node1 haproxy-1.7.8]# make TARGET=generic  gcc -Iinclude -Iebtree -Wall -O2 -g -fno-strict-aliasing -Wdeclaration-after-statement -fwrapv  -DTPROXY -DENABLE\_POLL  -DCONFIG\_HAPROXY\_VERSION=\"1.7.8\"  -DCONFIG\_HAPROXY\_DATE=\"2017/07/07\" \  -DBUILD\_TARGET='"generic"' \  -DBUILD\_ARCH='""' \  -DBUILD\_CPU='"generic"' \  -DBUILD\_CC='"gcc"' \  -DBUILD\_CFLAGS='"-O2 -g -fno-strict-aliasing -Wdeclaration-after-statement -fwrapv"' \  -DBUILD\_OPTIONS='""' \  -c -o src/haproxy.o src/haproxy.c  gcc -Iinclude -Iebtree -Wall -O2 -g -fno-strict-aliasing -Wdeclaration-after-statement -fwrapv...  ...  gcc -g -o haproxy src/haproxy.o src/base64.o src/protocol.o src/uri\_auth.o ... |

编译完目录下有名为“haproxy”的可执行文件。之后再/etc/profile中加入haproxy的路径，内容如下：

export PATH=$PATH:/opt/haproxy-1.7.8/haproxy

最后执行source /etc/profile让此环境变量生效。

### 配置HAProxy

HAProxy使用单一配置文件来定义所有属性，包括从前端IP到后端服务器。下面展示了用于3个RabbitMQ节点组成集群的负载均衡配置。这3个节点的IP地址分别为192.168.02、192.168.0.3、192.168.0.4，HAProxy运行在192.168.0.9这台机器上。

|  |
| --- |
| #全局配置  global  #日志输出配置，所有日志都记录在本机，通过local0输出  log 127.0.0.1 local0 info  #最大连接数  maxconn 4096  #改变当前的工作目录  chroot /opt/haproxy-1.7.8  #以指定的UID运行haproxy进程  uid 99  #以指定的GID运行haproxy进程  gid 99  #以守护进程方式运行haproxy #debug #quiet  daemon  #debug  #当前进程pid文件  pidfile /opt/haproxy-1.7.8/haproxy.pid  #默认配置  defaults  #应用全局的日志配置  log global  #默认的模式mode{tcp|http|health}  #tcp是4层，http是7层，health只返回OK  mode tcp  #日志类别tcplog  option tcplog  #不记录健康检查日志信息  option dontlognull  #3次失败则认为服务不可用  retries 3  #每个进程可用的最大连接数  maxconn 2000  #连接超时  timeout connect 5s  #客户端超时  timeout client 120s  #服务端超时  timeout server 120s  #绑定配置  listen rabbitmq\_cluster 5671  #配置TCP模式  mode tcp  #简单的轮询  balance roundrobin  #RabbitMQ集群节点配置  server rmq\_node1 192.168.0.2:5672 check inter 5000 rise 2 fall 3 weight 1  server rmq\_node2 192.168.0.3:5672 check inter 5000 rise 2 fall 3 weight 1  server rmq\_node3 192.168.0.4:5672 check inter 5000 rise 2 fall 3 weight 1  #haproxy监控页面地址  listen monitor :8100  mode http  option httplog  stats enable  stats uri /stats  stats refresh 5s |

在上面的配置中“listen rabbitmq\_cluster bind 192.168.0.9.5671”这里定义了客户端连接IP地址和端口号。这里配置的负载均衡算法是roundrobin，注意这里的roundrobin是加权轮询。和RabbitMQ最相关的是“ server rmq\_node1 192.168.0.2:5672 check inter 5000 rise 2 fall 3 weight 1”这种，它定义了RabbitMQ服务，每个RabbitMQ服务定义指令包含6个部分：

server <name>：定义RabbitMQ服务的内部标示，注意这里的“rmq\_node”是指包含有含义的字符串名称，不是指RabbitMQ的节点名称。

<ip>:<port>：定义RabbitMQ服务的连接的IP地址和端口号。

check inter <value>：定义了每隔多少毫秒检查RabbitMQ服务是否可用。

rise <value>：定义了RabbitMQ服务在发生故障之后，需要多少次健康检查才能被再次确认可用。

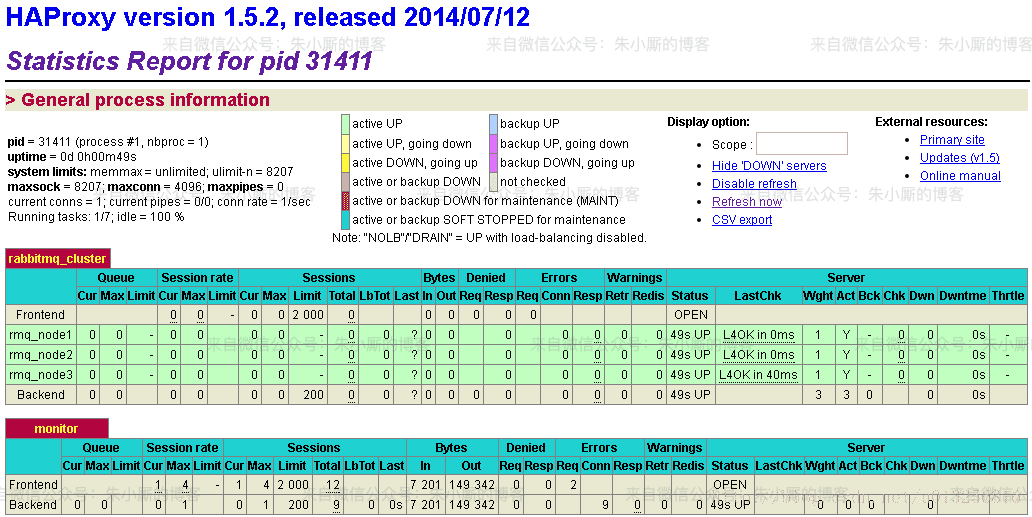
fall <value>：定义需要经历多少次失败的健康检查之后，HAProxy才会停止使用此RabbitMQ服务。

weight <value>：定义了当前RabbitMQ服务的权重。

最后一段配置定义的是HAProxy的数据统计页面。数据统计页面包含各个服务节点的状态、连接、负载等信息。在调用：

[root@node1 haproxy-1.7.8]# haproxy -f haproxy.cfg

运行HAProxy之后可以在浏览器上输入http://192.168.0.9:8100/stats来加载相关的页面，如下图所示：



## Keepalived+HAProxy

### 引入

试想下如果前面配置的HAProxy主机192.168.0.9突然宕机或者网卡失效，那么虽然RabbitMQ集群没有任何故障，但是对于外界的客户端来说所有的连接都会被断开，结果将是灾难性的。确保负载均衡服务的可靠性同样显得十分的重要。这里就引入Keepalived工具，它能够通过自身健康检查、资源接管功能做高可用（双机热备），实现故障转移。

Keepalived采用VRRP（Virtual Router Redundancy Protocol，虚拟路由冗余协议），以软件的形式实现服务器热备功能。通常情况下是将两台Linux服务器组成一个热备组（Master和Backup），同一时间热备组内只有一台主服务器Master提供服务，同时Master会虚拟出一个公用的虚拟IP地址，简称VIP。这个VIP只存在在Master上并对外提供服务。如果Keepalived检测到Master宕机或者服务故障，备份服务器Backup会自动接管VIP称为Master，Keepalived并将原Master从热备组中移除。当原Master恢复后，会自动加入到热备组，默认再抢占称为Master，起到故障转移的功能。

Keepalived工作在OSI模型中的第3层、第4层和第7层。

工作在第3层是指Keepalived会定期向热备组中的服务器发送一个ICMP数据包来判断某台服务器是否故障，如果故障则将这台服务器从热备组移除。

工作在第4层是指Keepalived以TCP端口的状态判断服务器是否故障，比如检测RabbitMQ的5672端口，如果故障则将这台服务器从热备组中移除。

工作在第7层是指Keepalived根据用户设定的策略（通常是一个自定义的检测脚本）判断服务器上的程序是否正常运行，如果故障将这台服务器从热备组移除。

### Keeplived

Vip:keepalived 虚拟出来的ip。

1、第一台haproxy会将这个虚拟的IP绑定到自己身上。



2、第一台haproxy挂机之后，另一台的haproxy感知到之后，会将这个vip绑定到自己的身上。



### Keepalived的安装

首先需要去Keepalived的官网下载Keepalived的安装文件，目前最新的版本为：keepalived-1.3.5.tar.gz，下载地址为<http://www.keepalived.org/download.html>。

将keepalived-1.3.5.tar.gz解压并安装，详细步骤如下：

|  |
| --- |
| [root@node1 ~]# tar zxvf keepalived-1.3.5.tar.gz  [root@node1 ~]# cd keepalived-1.3.5  [root@node1 keepalived-1.3.5]# ./configure --prefix=/opt/keepalived --with-init=SYSV  #注：(upstart|systemd|SYSV|SUSE|openrc) #根据你的系统选择对应的启动方式  [root@node1 keepalived-1.3.5]# make  [root@node1 keepalived-1.3.5]# make install |

之后将安装过后的Keepalived加入系统服务中，详细步骤如下（注意千万不要输错命令）：

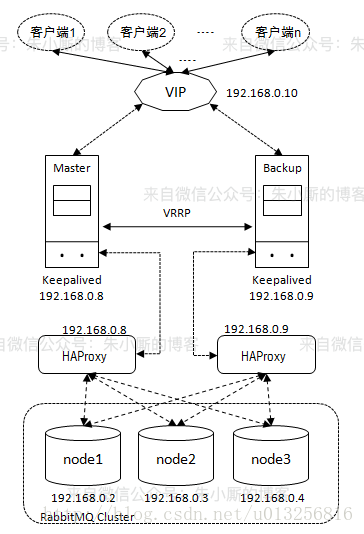
|  |
| --- |
| #复制启动脚本到/etc/init.d/下  [root@node1 ~]# cp /opt/keepalived/etc/rc.d/init.d/keepalived /etc/init.d/  [root@node1 ~]# cp /opt/keepalived/etc/sysconfig/keepalived /etc/sysconfig  [root@node1 ~]# cp /opt/keepalived/sbin/keepalived /usr/sbin/  [root@node1 ~]# chmod +x /etc/init.d/keepalived  [root@node1 ~]# chkconfig --add keepalived  [root@node1 ~]# chkconfig keepalived on  #Keepalived默认会读取/etc/keepalived/keepalived.conf配置文件  [root@node1 ~]# mkdir /etc/keepalived  [root@node1 ~]# cp /opt/keepalived/etc/keepalived/keepalived.conf /etc/keepalived/ |

执行完之后就可以使用如下：

|  |
| --- |
| service keepalived restart  service keepalived start  service keepalived stop  service keepalived status |

这4个命令来重启、启动、关闭和查看keepalived状态。

在安装的时候我们已经创建了/etc/keepalived目录，并将keepalived.conf配置文件拷贝到此目录下，如此Keepalived便可以读取这个默认的配置文件了。如果要将Keepalived与前面的HAProxy服务结合起来需要更改/etc/keepalived/keepalived.conf这个配置文件，在此之前先来看看本次配置需要完成的详情及目标。



如图所示，两台Keepalived服务器之间通过VRRP进行交互，对外部虚拟出一个VIP为192.168.0.10。Keepalived与HAProxy部署在同一台机器上，两个Keepalived服务实例匹配两个HAProxy服务实例，这样通过Keeaplived实现HAProxy的双机热备。所以在上一小节的192.168.0.9的基础之上，还要再部署一台HAProxy服务，IP地址为192.168.0.8。整条调用链路为：客户端通过VIP建立通信链路；通信链路通过Keeaplived的Master节点路由到对应的HAProxy之上；HAProxy通过负载均衡算法将负载分发到集群中的各个节点之上。正常情况下客户端的连接通过图中左侧部分进行负载分发。当Keepalived的Master节点挂掉或者HAProxy挂掉无法恢复，那么Backup提升为Master，客户端的连接通过图中右侧部分进行负载分发。如果你追求要更高的可靠性，可以加入多个Backup角色的Keepalived节点来实现一主多从的多机热备。当然这样会提升硬件资源的成本，该如何抉择需要更细致的考恒，一般情况下双机热备的配备已足够满足应用需求。

接下来我们要修改/etc/keepalived/keepalived.conf文件，在Keepalived的Master上配置详情如下：

|  |
| --- |
| #Keepalived配置文件  global\_defs {  router\_id NodeA #路由ID, 主备的ID不能相同  }  #自定义监控脚本  vrrp\_script chk\_haproxy {  script "/etc/keepalived/check\_haproxy.sh"  interval 5  weight 2  }  vrrp\_instance VI\_1 {  state MASTER #Keepalived的角色。Master表示主服务器，从服务器设置为BACKUP  interface eth0 #指定监测网卡  virtual\_router\_id 1  priority 100 #优先级，BACKUP机器上的优先级要小于这个值  advert\_int 1 #设置主备之间的检查时间，单位为s  authentication { #定义验证类型和密码  auth\_type PASS  auth\_pass root123  }  track\_script {  chk\_haproxy  }  virtual\_ipaddress { #VIP地址，可以设置多个：  192.168.0.10  }  } |

Backup中的配置大致和Master中的相同，不过需要修改global\_defs{}的router\_id，比如置为NodeB；其次要修改vrrp\_instance VI\_1{}中的state为BACKUP；最后要将priority设置为小于100的值。注意Master和Backup中的virtual\_router\_id要保持一致。下面简要的展示下Backup的配置：

|  |
| --- |
| global\_defs {  router\_id NodeB  }  vrrp\_script chk\_haproxy {  ...  }  vrrp\_instance VI\_1 {  state BACKUP  ...  priority 50  ...  } |

为了防止HAProxy服务挂了，但是Keepalived却还在正常工作而没有切换到Backup上，所以这里需要编写一个脚本来检测HAProxy服务的状态。当HAProxy服务挂掉之后该脚本会自动重启HAProxy的服务，如果不成功则关闭Keepalived服务，如此便可以切换到Backup继续工作。这个脚本就对应了上面配置中vrrp\_script chk\_haproxy{}的script对应的值，/etc/keepalived/check\_haproxy.sh的内容如代码清单所示。

|  |
| --- |
| #!/bin/bash  if [ $(ps -C haproxy --no-header | wc -l) -eq 0 ];then  haproxy -f /opt/haproxy-1.7.8/haproxy.cfg  fi  sleep 2  if [ $(ps -C haproxy --no-header | wc -l) -eq 0 ];then  service keepalived stop  fi |

如此配置好之后，使用service keepalived start命令启动192.168.0.8和192.168.0.9中的Keepalived服务即可。之后客户端的应用可以通过192.168.0.10这个IP地址来接通RabbitMQ服务。

##查看Keepalived的运行情况

可以通过tail -f /var/log/messages -n 200命令查看相应的Keepalived日志输出。Master启动日志如下：

|  |
| --- |
| Oct 4 23:01:51 node1 Keepalived[30553]: Starting Keepalived v1.3.5 (03/19,2017), git commit v1.3.5-6-g6fa32f2  Oct 4 23:01:51 node1 Keepalived[30553]: Unable to resolve default script username 'keepalived\_script' - ignoring  Oct 4 23:01:51 node1 Keepalived[30553]: Opening file '/etc/keepalived/keepalived.conf'.  Oct 4 23:01:51 node1 Keepalived[30554]: Starting Healthcheck child process, pid=30555  Oct 4 23:01:51 node1 Keepalived[30554]: Starting VRRP child process, pid=30556  Oct 4 23:01:51 node1 Keepalived\_healthcheckers[30555]: Opening file '/etc/keepalived/keepalived.conf'.  Oct 4 23:01:51 node1 Keepalived\_vrrp[30556]: Registering Kernel netlink reflector  Oct 4 23:01:51 node1 Keepalived\_vrrp[30556]: Registering Kernel netlink command channel  Oct 4 23:01:51 node1 Keepalived\_vrrp[30556]: Registering gratuitous ARP shared channel  Oct 4 23:01:51 node1 Keepalived\_vrrp[30556]: Opening file '/etc/keepalived/keepalived.conf'.  Oct 4 23:01:51 node1 Keepalived\_vrrp[30556]: VRRP\_Instance(VI\_1) removing protocol VIPs.  Oct 4 23:01:51 node1 Keepalived\_vrrp[30556]: SECURITY VIOLATION - scripts are being executed but script\_security not enabled.  Oct 4 23:01:51 node1 Keepalived\_vrrp[30556]: Using LinkWatch kernel netlink reflector...  Oct 4 23:01:51 node1 Keepalived\_vrrp[30556]: VRRP sockpool: [ifindex(2), proto(112), unicast(0), fd(10,11)]  Oct 4 23:01:51 node1 Keepalived\_vrrp[30556]: VRRP\_Instance(VI\_1) Transition to MASTER STATE  Oct 4 23:01:52 node1 Keepalived\_vrrp[30556]: VRRP\_Instance(VI\_1) Entering MASTER STATE  Oct 4 23:01:52 node1 Keepalived\_vrrp[30556]: VRRP\_Instance(VI\_1) setting protocol VIPs. |

Master启动之后可以通过ip add show命令查看添加的VIP（加粗部分，Backup节点是没有VIP的）：

|  |
| --- |
| [root@node1 ~]# ip add show  1: lo: <LOOPBACK,UP,LOWER\_UP> mtu 65536 qdisc noqueue state UNKNOWN  link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00  inet 127.0.0.1/8 scope host lo  inet6 ::1/128 scope host  valid\_lft forever preferred\_lft forever  2: eth0: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1450 qdisc pfifo\_fast state UP qlen 1000  link/ether fa:16:3e:5e:7a:f7 brd ff:ff:ff:ff:ff:ff  inet 192.168.0.8/18 brd xx.xx.255.255 scope global eth0  inet 192.168.0.10/32 scope global eth0  inet6 fe80::f816:3eff:fe5e:7af7/64 scope link  valid\_lft forever preferred\_lft forever |

在Master节点执行 service keepalived stop模拟异常关闭的情况，观察Master的日志：

|  |
| --- |
| Oct 4 22:58:32 node1 Keepalived[27609]: Stopping  Oct 4 22:58:32 node1 Keepalived\_vrrp[27611]: VRRP\_Instance(VI\_1) sent 0 priority  Oct 4 22:58:32 node1 Keepalived\_vrrp[27611]: VRRP\_Instance(VI\_1) removing protocol VIPs.  Oct 4 22:58:32 node1 Keepalived\_healthcheckers[27610]: Stopped  Oct 4 22:58:33 node1 Keepalived\_vrrp[27611]: Stopped  Oct 4 22:58:33 node1 Keepalived[27609]: Stopped Keepalived v1.3.5 (03/19,2017), git commit v1.3.5-6-g6fa32f2  Oct 4 22:58:34 node1 ntpd[1313]: Deleting interface #13 eth0, 192.168.0.10#123, interface stats: received=0, sent=0, dropped=0, active\_time=532 secs  Oct 4 22:58:34 node1 ntpd[1313]: peers refreshed |

对应的Master上的VIP也会消失：

|  |
| --- |
| [root@node1 ~]# ip add show  1: lo: <LOOPBACK,UP,LOWER\_UP> mtu 65536 qdisc noqueue state UNKNOWN  link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00  inet 127.0.0.1/8 scope host lo  inet6 ::1/128 scope host  valid\_lft forever preferred\_lft forever  2: eth0: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1450 qdisc pfifo\_fast state UP qlen 1000  link/ether fa:16:3e:5e:7a:f7 brd ff:ff:ff:ff:ff:ff  inet 192.168.0.8/18 brd xx.xx.255.255 scope global eth0  inet6 fe80::f816:3eff:fe5e:7af7/64 scope link  valid\_lft forever preferred\_lft forever |

Master关闭后，Backup会提升为新的Master，对应的日志为：

|  |
| --- |
| Oct 4 22:58:15 node2 Keepalived\_vrrp[2352]: VRRP\_Instance(VI\_1) Transition to MASTER STATE  Oct 4 22:58:16 node2 Keepalived\_vrrp[2352]: VRRP\_Instance(VI\_1) Entering MASTER STATE  Oct 4 22:58:16 node2 Keepalived\_vrrp[2352]: VRRP\_Instance(VI\_1) setting protocol VIPs. |

可以看到新的Master节点上虚拟出了VIP如下所示：

|  |
| --- |
| [root@node2 ~]# ip add show  1: lo: <LOOPBACK,UP,LOWER\_UP> mtu 65536 qdisc noqueue state UNKNOWN  link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00  inet 127.0.0.1/8 scope host lo  inet6 ::1/128 scope host  valid\_lft forever preferred\_lft forever  2: eth0: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1450 qdisc pfifo\_fast state UP qlen 1000  link/ether fa:16:3e:23:ac:ec brd ff:ff:ff:ff:ff:ff  inet 192.168.0.9/18 brd xx.xx.255.255 scope global eth0  inet 192.168.0.10/32 scope global eth0  inet6 fe80::f816:3eff:fe23:acec/64 scope link  valid\_lft forever preferred\_lft forever |

# Rabbmq集群

## Rabbitmq集群

### 引入

### 集群要解决的问题

可拓展

数据冗余:rabbitmq镜像队列，将节点的数据复制到其他数据。

高可用

### 集群造成的问题

### 高可用负载均衡

## 安装

### 安装环境

sudo apt-get install erlang-nox

sudo apt-get update

### 下载包

sudo apt-get install rabbitmq-server

启动、停止、重启、状态rabbitMq命令

sudo rabbitmq-server start

sudo rabbitmq-server stop

sudo rabbitmq-server restart

sudo rabbitmqctl status

### 创建用户

添加admin用户，密码设置为123456。

sudo rabbitmqctl add\_user admin 123456

赋予权限

sudo rabbitmqctl set\_user\_tags admin administrator

赋予virtual host中所有资源的配置、写、读权限以便管理其中的资源

sudo rabbitmqctl set\_permissions -p / admin '.\*' '.\*' '.\*'

### 访问管控台

命令:sudo rabbitmq-plugins enable rabbitmq\_management（先定位到rabbitmq安装目录）

地址: http:// 192.168.205.129:15672/

### 修改erlangcookie

将一台机器的erlangcookie 覆盖其他的两台

chmod 777 /var/lib/rabbitmq/.erlang.cookie

scp /var/lib/rabbitmq/.erlang.cookie [root@mq02:/var/lib/rabbitmq/.erlang.cookie](mailto:root@mq02:/var/lib/rabbitmq/.erlang.cookie)

改回文件权限

chmod 400 /var/lib/rabbitmq/.erlang.cookie

### 启动

启动rabbitmq:

sudo rabbitmq-server start

停掉:

rabbitmqctl stop\_app

### 集群命令

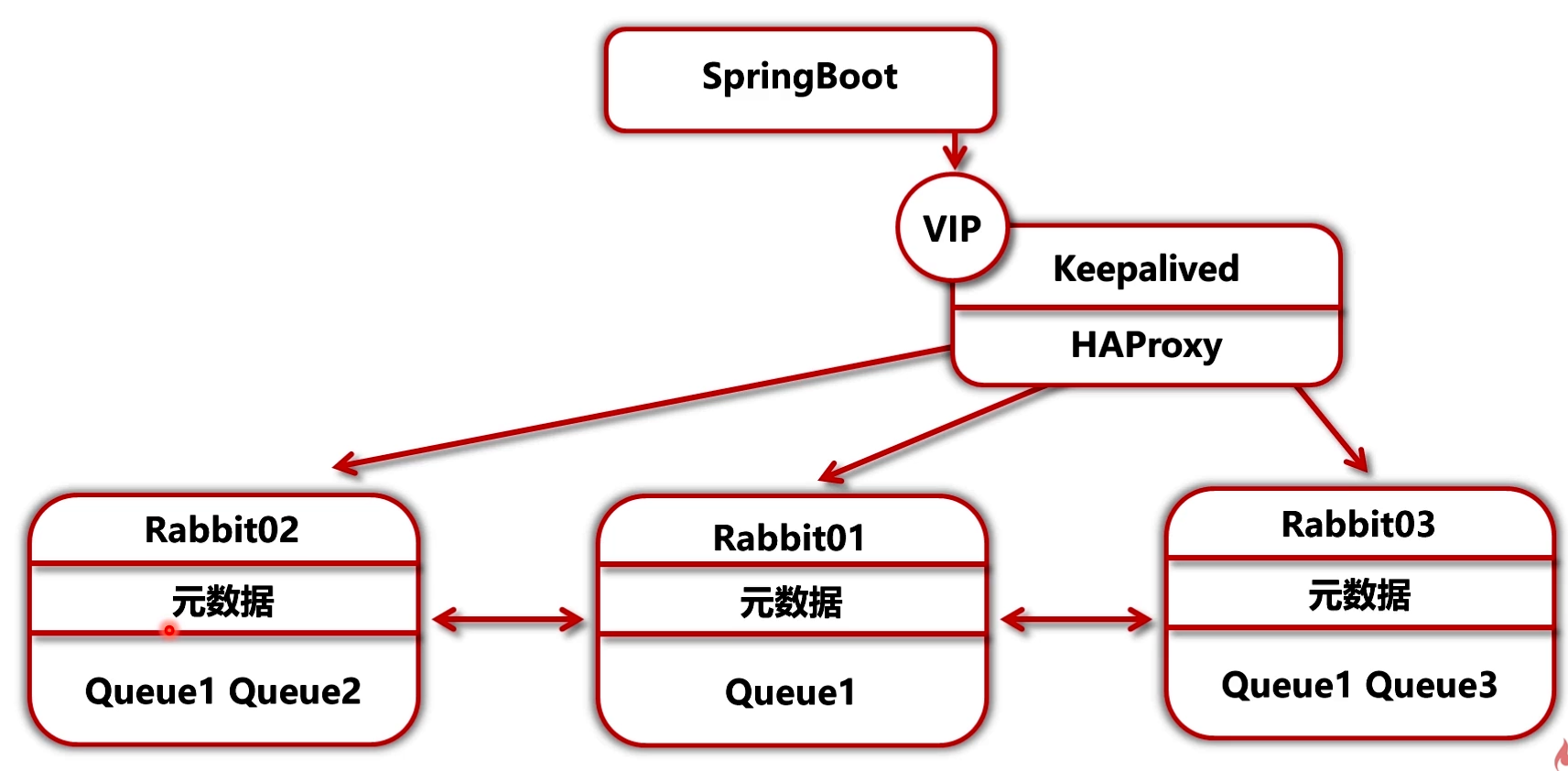
Mq02:rabbitmqctl join\_cluster rabbit@mq01

### 设置镜像队列

镜像队列配置

# Rabbmq集群(docker)

## 引入



如果一台机器宕机之之后需要手动的去做修复。

在传统以物理机/虚拟机为基础的架构中,服务宕机往往需要人工处理

随着容器技术的发展,容器编排框架可以很好的解决高可用问题

K8S已经成为容器编排的事实标准,能够承载RabbitMQ集群

## Docker部署集群(伪集群)

### 安装docker

依赖包:sudo yum install yum-utils device-mapper-persistent-data lvm2 -y

添加仓库:sudo yum-config-manager --add-repo <https://download.docker.com/linux/centos/docker-ce.repo>

安装docker:sudo yum install docker-ce

### Docker命令

开启启动docker:systemctl enable docker

启动docker:systemctl start docker

查看版本: docker -v

### 启动rabbitmq

创建实例: docker run -d --rm --name rabbitmq -p 5672:5672 -p15672:15672 rabbitmq:3-management

### Docker-compose.yml

1、添加EPEL源在Shell中运行以下命令，安装EPEL源：

yum install -y epel-release

2、安装python-pip在Shell中运行以下命令，安装Python-pip软件包：

yum install -y python-pip

yum install -y python3-pip#python3安装pip3!!!

3、安装docker-compose，在Shell中运行以下命令，通过pip安装docker-compose：

pip3 install docker-compose --default-timeout=100

4、查看安装版

docker-compose version

5、启动

docker-compose up -d

6、查看

Docker ps

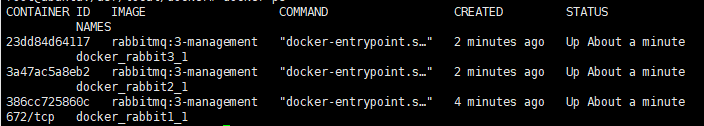
### 配置文件

|  |
| --- |
| version: "2.0"  services:  rabbit1:  image: rabbitmq:3-management  hostname: rabbit1  ports:  - 5672:5672  - 15672:15672  environment:  - RABBITMQ\_DEFAULT\_USER=root  - RABBITMQ\_DEFAULT\_pass=123456  - RABBITMQ\_ERLANG\_COOKIE='123456'  rabbit2:  image: rabbitmq:3-management  hostname: rabbit2  ports:  - 5673:5672  environment:  - RABBITMQ\_ERLANG\_COOKIE='123456'  links:  - rabbit1  rabbit3:  image: rabbitmq:3-management  hostname: rabbit3  ports:  - 5674:5672  environment:  - RABBITMQ\_ERLANG\_COOKIE='123456'  links:  - rabbit1 |

### 配置集群

启动: docker-compose up -d

查看:docker ps



将2、3节点接入到1的就集群节点中。

进入容器:docker exec -it docker\_rabbit2\_1 bash’

停掉app:rabbitmqctl stop\_app

加入节点: rabbitmqctl join\_cluster rabbit@rabbit1

启动节点: rabbitmqctl start\_app

退出:exit

添加用户

添加admin用户，密码设置为123456。

sudo rabbitmqctl add\_user admin 123456

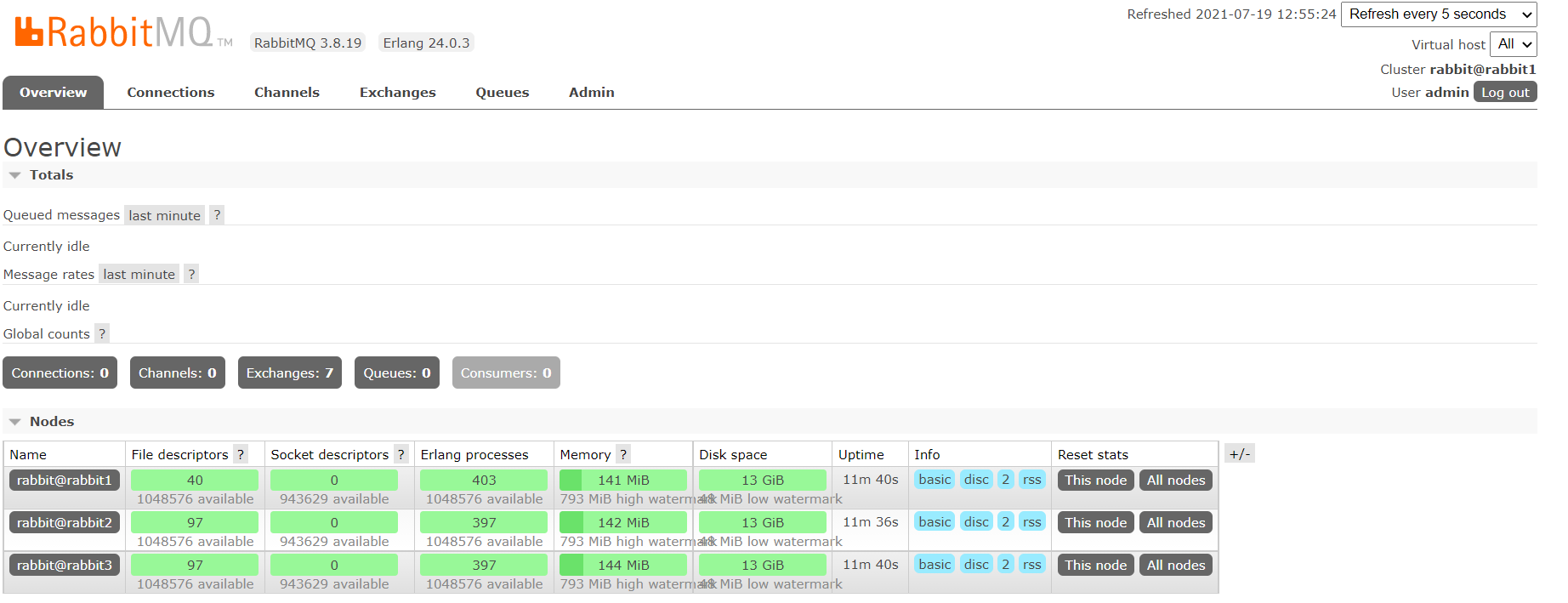
赋予权限

sudo rabbitmqctl set\_user\_tags admin administrator

赋予virtual host中所有资源的配置、写、读权限以便管理其中的资源

sudo rabbitmqctl set\_permissions -p / admin '.\*' '.\*' '.\*'

查看: http://192.168.205.131:15672/#/



## 集群(K8S)

# Rabbit事务框架

## 分布式事务

一致性、可用性、分区容忍性不可能三者兼顾

一致性(Consistency)

可用性(Availability)

分区容忍性(Partition tolerance)

### Base理论

,由于CAP无法同时满足,基于工程实际,提出了BASE理论

Basically Available (基本可用)

Soft state (软状态)

最终一致性: Eventually consistent (最终一致性)

ACID往往针对传统本地事务,分布式事务无法满足原子性和

隔离性,需要舍弃传统ACID理论

基于BASE理论,业务状态不需要在微服务系统内强一致，基于BASE理论,订单状态要做到最终一致性即可

为了做到最终一致性,要保证消息不丢失,发送-处理的流程要有重试机制,重试多次失败后要有告警

## 框架设计

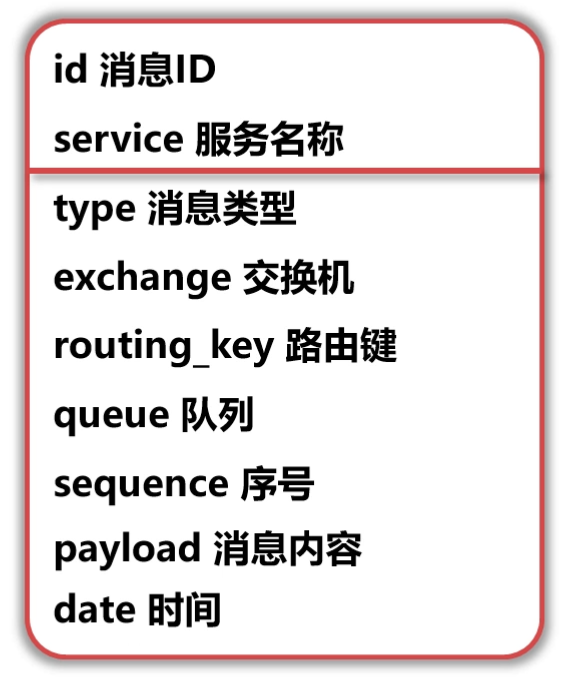
### 框架考虑点

1、发送失败重试

2、消费失败重试

3、死信告警

### 数据表设计



### 消息发送失败重试

1、发送消息前消息持久化

2、发送成功时删除消息

3、定时巡检未发送成功消息、重试发送



死信队列: